1 User’s Guide

1 Introduction

2 Configuration Guide
  2.1 Installing
  2.2 Customizing matplotlib
  2.3 Using matplotlib in a python shell

3 Beginner’s Guide
  3.1 Pyplot tutorial
  3.2 Customizing plots with style sheets
  3.3 Interactive navigation
  3.4 Working with text
  3.5 Image tutorial
  3.6 Legend guide
  3.7 Annotating Axes
  3.8 Screenshots
  3.9 Choosing Colormaps

4 Advanced Guide
  4.1 Artist tutorial
  4.2 Customizing Location of Subplot Using GridSpec
  4.3 Tight Layout guide
  4.4 Event handling and picking
  4.5 Transformations Tutorial
  4.6 Path Tutorial
  4.7 Path effects guide
  4.8 Our Favorite Recipes

5 What’s new in matplotlib
  5.1 new in matplotlib-1.4
  5.2 new in matplotlib-1.3
  5.3 new in matplotlib 1.2.2
  5.4 new in matplotlib-1.2
  5.5 new in matplotlib-1.1
  5.6 new in matplotlib-1.0
22.2 Organization of matplotlib’s documentation .............................................. 427
22.3 Formatting ................................................................................................. 429
22.4 Figures ....................................................................................................... 431
22.5 Referring to mpl documents ...................................................................... 434
22.6 Internal section references ........................................................................ 435
22.7 Section names, etc ..................................................................................... 435
22.8 Inheritance diagrams ............................................................................... 435
22.9 Emacs helpers ......................................................................................... 436

23 Doing a matplotlib release ........................................................................... 439
  23.1 Testing ..................................................................................................... 439
  23.2 Branching ............................................................................................... 439
  23.3 Packaging ............................................................................................... 440
  23.4 Posting files ........................................................................................... 440
  23.5 Update PyPI ........................................................................................... 441
  23.6 Documentation updates ......................................................................... 442
  23.7 Announcing ........................................................................................... 442

24 Working with transformations ....................................................................... 443
  24.1 matplotlib.transforms ............................................................................ 443

25 Adding new scales and projections to matplotlib ......................................... 465
  25.1 Creating a new scale ............................................................................... 465
  25.2 Creating a new projection ....................................................................... 466
  25.3 API documentation ............................................................................... 467

V Matplotlib AxesGrid Toolkit ........................................................................ 477

26 Overview of AxesGrid toolkit ..................................................................... 481
  26.1 What is AxesGrid toolkit? ....................................................................... 481
  26.2 AXES_GRID1 ......................................................................................... 483
  26.3 AXISARTIST ......................................................................................... 497

27 The Matplotlib AxesGrid Toolkit User’s Guide ........................................... 503
  27.1 AxesDivider ......................................................................................... 503
  27.2 AXISARTIST namespace .................................................................... 506

28 The Matplotlib AxesGrid Toolkit API ......................................................... 519
  28.1 mpl_toolkits.axes_grid.axes_size ......................................................... 519
  28.2 mpl_toolkits.axes_grid.axes_divider ..................................................... 520
  28.3 mpl_toolkits.axes_grid.axes_grid ......................................................... 524
  28.4 mpl_toolkits.axes_grid.axis_artist ....................................................... 525

VI mplot3d ...................................................................................................... 529

29 Matplotlib mplot3d toolkit ......................................................................... 531
  29.1 mplot3d tutorial ................................................................................... 531
34.21 Changes for 0.81 ................................................................. 727
34.22 Changes for 0.80 ................................................................. 728
34.23 Changes for 0.73 ................................................................. 728
34.24 Changes for 0.72 ................................................................. 728
34.25 Changes for 0.71 ................................................................. 729
34.26 Changes for 0.70 ................................................................. 729
34.27 Changes for 0.65.1 .............................................................. 730
34.28 Changes for 0.65 ................................................................. 730
34.29 Changes for 0.63 ................................................................. 730
34.30 Changes for 0.61 ................................................................. 731
34.31 Changes for 0.60 ................................................................. 731
34.32 Changes for 0.54.3 .............................................................. 731
34.33 Changes for 0.54 ................................................................. 732
34.34 Changes for 0.50 ................................................................. 735
34.35 Changes for 0.42 ................................................................. 736
34.36 Changes for 0.40 ................................................................. 737

35 The top level matplotlib module .............................................. 739

36 afm (Adobe Font Metrics interface) ........................................ 743
  36.1 matplotlib.afm ............................................................... 743

37 animation ................................................................. 747
  37.1 matplotlib.animation ...................................................... 747

38 artists ................................................................. 755
  38.1 matplotlib.artist ............................................................ 757

39 axes ................................................................. 767
  39.1 matplotlib.axes ............................................................. 767

40 axis ................................................................. 953
  40.1 matplotlib.axis .............................................................. 953

41 backends ............................................................... 963
  41.1 matplotlib.backend_bases .................................................. 963
  41.2 matplotlib.backends.backend_gtkagg .................................... 981
  41.3 matplotlib.backends.backend_qt4agg ................................... 981
  41.4 matplotlib.backends.backend_wxagg ................................... 982
  41.5 matplotlib.backends.backend_pdf ...................................... 983

42 cbook ............................................................... 987
  42.1 matplotlib.cbook ........................................................... 987

43 cm (colormap) ............................................................ 1001
  43.1 matplotlib.cm ............................................................. 1001

44 collections ............................................................ 1005
  44.1 matplotlib.collections .................................................. 1005
74.8 animation example code: histogram.py ........................................ 1636
74.9 animation example code: moviewriter.py ................................ 1637
74.10 animation example code: rain.py ............................................. 1638
74.11 animation example code: random_data.py .............................. 1639
74.12 animation example code: simple_3danim.py ............................ 1640
74.13 animation example code: simple_anim.py ............................... 1641
74.14 animation example code: strip_chart_demo.py ......................... 1642
74.15 animation example code: subplots.py .................................... 1643
74.16 animation example code: unchained.py .................................. 1645

75 api Examples ................................................................. 1647
75.1 api example code: agg_oo.py ................................................. 1647
75.2 api example code: barchart_demo.py ..................................... 1648
75.3 api example code: bbox_intersect.py .................................... 1649
75.4 api example code: collections_demo.py .................................. 1650
75.5 api example code: colorbar_only.py .................................... 1653
75.6 api example code: compound_path.py ................................... 1655
75.7 api example code: custom_projection_example.py .................... 1657
75.8 api example code: custom_scale_example.py ............................. 1667
75.9 api example code: date_demo.py .......................................... 1671
75.10 api example code: date_index_formatter.py ......................... 1673
75.11 api example code: demo_affine_image.py .............................. 1675
75.12 api example code: donut_demo.py ................................. 1677
75.13 api example code: engineering_formatter.py .................... 1679
75.14 api example code: font_family_rc.py .................................. 1680
75.15 api example code: font_file.py ......................................... 1680
75.16 api example code: histogram_path_demo.py .......................... 1682
75.17 api example code: image_zcoord.py .................................. 1684
75.18 api example code: joinstyle.py ....................................... 1685
75.19 api example code: legend_demo.py .................................... 1686
75.20 api example code: line_with_text.py .................................. 1687
75.21 api example code: log2.py .............................................. 1689
75.22 api example code: mathtext_asarray.py .............................. 1691
75.23 api example code: patch_collection.py ................................ 1692
75.24 api example code: power_norm_demo.py ............................. 1694
75.25 api example code: quad bezier.py .................................... 1695
75.26 api example code: radar_chart.py ..................................... 1696
75.27 api example code: sankey_demo basics.py ............................ 1701
75.28 api example code: sankey_demo_links.py ........................... 1705
75.29 api example code: sankey_demo_old.py .............................. 1707
75.30 api example code: sankey_demo rankine.py .......................... 1712
75.31 api example code: scatter_piecharts.py .......................... 1714
75.32 api example code: skewt.py ............................................. 1716
75.33 api example code: span_regions.py ................................. 1722
75.34 api example code: two_scales.py ...................................... 1723
75.35 api example code: unicode_minus.py .................................. 1725
75.36 api example code: watermark_image.py ............................. 1726
75.37 api example code: watermark_text.py ................................ 1727
76 axes_grid Examples

76.1 axes_grid example code: demo_axes_divider.py .................................................. 1729
76.2 axes_grid example code: demo_axes_grid.py ......................................................... 1732
76.3 axes_grid example code: demo_axes_grid2.py ..................................................... 1735
76.4 axes_grid example code: demo_axes_hbox_divider.py ........................................... 1738
76.5 axes_grid example code: demo_axes_rgb.py .......................................................... 1740
76.6 axes_grid example code: demo_axisline_style.py .................................................. 1743
76.7 axes_grid example code: demo_colorbar_with_inset_locator.py ............................... 1744
76.8 axes_grid example code: demo_curvelinear_grid.py .............................................. 1745
76.9 axes_grid example code: demo_curvelinear_grid2.py ........................................... 1748
76.10 axes_grid example code: demo_edge_colorbar.py ................................................ 1750
76.11 axes_grid example code: demo_floating_axes.py ................................................. 1752
76.12 axes_grid example code: demo_floating_axis.py ............................................... 1756
76.13 axes_grid example code: demo_imagegrid_aspect.py .......................................... 1758
76.14 axes_grid example code: demo_parasite_axes2.py .............................................. 1759
76.15 axes_grid example code: inset_locator_demo.py ............................................... 1760
76.16 axes_grid example code: inset_locator_demo2.py .............................................. 1762
76.17 axes_grid example code: make_room_for_ylabel_using_axesgrid.py ...................... 1764
76.18 axes_grid example code: parasite_simple2.py ................................................... 1768
76.19 axes_grid example code: scatter_hist.py .......................................................... 1770
76.20 axes_grid example code: simple_anchored_artists.py ........................................ 1772
76.21 axes_grid example code: simple_axesgrid.py .................................................... 1774
76.22 axes_grid example code: simple_axesgrid2.py ................................................... 1775
76.23 axes_grid example code: simple_axisline4.py ................................................. 1776

77 color Examples

77.1 color example code: color_cycle_demo.py .......................................................... 1779
77.2 color example code: colormaps_reference.py ...................................................... 1781
77.3 color example code: named_colors.py .................................................................. 1787

78 event_handling Examples

78.1 event_handling example code: close_event.py ....................................................... 1791
78.2 event_handling example code: data_browser.py .................................................... 1791
78.3 event_handling example code: figure_axes_enter_leave.py .................................. 1793
78.4 event_handling example code: idle_and_timeout.py .............................................. 1794
78.5 event_handling example code: keypress_demo.py ................................................ 1795
78.6 event_handling example code: lasso_demo.py ....................................................... 1795
78.7 event_handling example code: legend_picking.py ................................................ 1797
78.8 event_handling example code: looking_glass.py ................................................... 1798
78.9 event_handling example code: path_editor.py ...................................................... 1799
78.10 event_handling example code: pick_event_demo.py ............................................. 1802
78.11 event_handling example code: pick_event_demo2.py .......................................... 1805
78.12 event_handling example code: pipong.py ............................................................ 1806
78.13 event_handling example code: poly_editor.py ..................................................... 1811
78.14 event_handling example code: pong_gtk.py ........................................................ 1814
78.15 event_handling example code: resample.py ........................................................ 1815
78.16 event_handling example code: test_mouseclicks.py ............................................. 1816
78.17 event_handling example code: timers.py ............................................................ 1817
82.9  mplot3d example code: lines3d_demo.py .................................................. 1877
82.10 mplot3d example code: lorenz_attractor.py .............................................. 1878
82.11 mplot3d example code: mixed_subplots_demo.py ........................................ 1880
82.12 mplot3d example code: offset_demo.py ..................................................... 1882
82.13 mplot3d example code: pathpatch3d_demo.py ............................................. 1883
82.14 mplot3d example code: polys3d_demo.py .................................................... 1885
82.15 mplot3d example code: quiver3d_demo.py ................................................. 1886
82.16 mplot3d example code: rotate_axes3d_demo.py .......................................... 1887
82.17 mplot3d example code: scatter3d_demo.py ............................................... 1888
82.18 mplot3d example code: subplot3d_demo.py ............................................... 1889
82.19 mplot3d example code: surface3d_demo.py ................................................. 1890
82.20 mplot3d example code: surface3d_demo2.py .............................................. 1892
82.21 mplot3d example code: surface3d_demo3.py .............................................. 1893
82.22 mplot3d example code: surface3d_radial_demo.py ..................................... 1894
82.23 mplot3d example code: text3d_demo.py ..................................................... 1895
82.24 mplot3d example code: tricontour3d_demo.py ............................................ 1897
82.25 mplot3d example code: tricontourf3d_demo.py .......................................... 1898
82.26 mplot3d example code: trisurf3d_demo.py ............................................... 1900
82.27 mplot3d example code: trisurf3d_demo2.py .............................................. 1901
82.28 mplot3d example code: wire3d_animation_demo.py ..................................... 1904
82.29 mplot3d example code: wire3d_demo.py ..................................................... 1905

83  pie_and_polar_charts Examples  ................................................................. 1907
83.1  pie_and_polar_charts example code: pie_demo_features.py ............................ 1907
83.2  pie_and_polar_charts example code: polar_bar_demo.py ............................... 1909
83.3  pie_and_polar_charts example code: polar_scatter_demo.py ........................... 1910

84  pylab_examples Examples  ............................................................................. 1913
84.1  pylab_examples example code: accented_text.py ......................................... 1913
84.2  pylab_examples example code: agg_buffer_to_array.py ............................... 1914
84.3  pylab_examples example code: alignment_test.py ......................................... 1916
84.4  pylab_examples example code: anchored_artists.py ..................................... 1918
84.5  pylab_examples example code: animation_demo.py ....................................... 1921
84.6  pylab_examples example code: annotation_demo.py ....................................... 1922
84.7  pylab_examples example code: annotation_demo2.py .................................... 1927
84.8  pylab_examples example code: annotation_demo3.py .................................... 1932
84.9  pylab_examples example code: anscombe.py ................................................. 1935
84.10 pylab_examples example code: arctest.py ................................................. 1937
84.11 pylab_examples example code: arrow_demo.py .............................................. 1938
84.12 pylab_examples example code: arrow_simple_demo.py ................................... 1945
84.13 pylab_examples example code: aspect_loglog.py ......................................... 1946
84.14 pylab_examples example code: axes_demo.py ................................................. 1947
84.15 pylab_examples example code: axes_props.py .............................................. 1948
84.16 pylab_examples example code: axes_zoom_effect.py .................................... 1950
84.17 pylab_examples example code: axhspan_demo.py .......................................... 1953
84.18 pylab_examples example code: axis_equal_demo.py ..................................... 1954
84.19 pylab_examples example code: bar_stacked.py ............................................ 1956
84.20 pylab_examples example code: barb_demo.py ............................................... 1957
84.21 pylab_examples example code: barchart_demo2.py .......................... 1960
84.22 pylab_examples example code: barchart_demo2.py .......................... 1962
84.23 pylab_examples example code: barcode_demo.py ............................ 1965
84.24 pylab_examples example code: boxplot_demo.py ........................... 1966
84.25 pylab_examples example code: boxplot_demo2.py .......................... 1974
84.26 pylab_examples example code: boxplot_demo3.py .......................... 1977
84.27 pylab_examples example code: break.py ..................................... 1979
84.28 pylab_examples example code: broken_axis.py ............................. 1980
84.29 pylab_examples example code: broken_barh.py ............................. 1982
84.30 pylab_examples example code: centered_ticklabels.py ................. 1983
84.31 pylab_examples example code: clippedline.py .............................. 1985
84.32 pylab_examples example code: cohere_demo.py ............................ 1987
84.33 pylab_examples example code: color_by_value.py .......................... 1988
84.34 pylab_examples example code: color_demo.py ............................... 1989
84.35 pylab_examples example code: colorbar_tick_labelling_demo.py ........ 1991
84.36 pylab_examples example code: colours.py ................................. 1993
84.37 pylab_examples example code: contour_demo.py ............................ 1994
84.38 pylab_examples example code: contour_image.py ........................... 2002
84.39 pylab_examples example code: contour_label_demo.py .................... 2005
84.40 pylab_examples example code: contourf_demo.py ........................... 2009
84.41 pylab_examples example code: contourf_hatching.py ...................... 2014
84.42 pylab_examples example code: contourf_log.py ............................. 2017
84.43 pylab_examples example code: coords_demo.py ............................. 2019
84.44 pylab_examples example code: coords_report.py ......................... 2020
84.45 pylab_examples example code: csd_demo.py ................................ 2021
84.46 pylab_examples example code: cursor_demo.py ............................. 2022
84.47 pylab_examples example code: custom_cmap.py ............................. 2026
84.48 pylab_examples example code: custom_figure_class.py .................. 2031
84.49 pylab_examples example code: custom_ticker1.py ......................... 2032
84.50 pylab_examples example code: customize_rc.py ............................ 2033
84.51 pylab_examples example code: dashpointlabel.py ......................... 2035
84.52 pylab_examples example code: data_helper.py ............................. 2036
84.53 pylab_examples example code: date_demo1.py ............................. 2038
84.54 pylab_examples example code: date_demo2.py ............................. 2040
84.55 pylab_examples example code: date_demo_convert.py ..................... 2042
84.56 pylab_examples example code: date_demo_rrule.py ....................... 2043
84.57 pylab_examples example code: date_index_formatter.py ................. 2045
84.58 pylab_examples example code: demo_agg_filter.py ....................... 2047
84.59 pylab_examples example code: demo_annotation_box.py .................. 2054
84.60 pylab_examples example code: demo_bboximage.py ....................... 2057
84.61 pylab_examples example code: demo_ribbon_box.py ....................... 2059
84.62 pylab_examples example code: demo_text_path.py ........................ 2062
84.63 pylab_examples example code: demo_text_rotation_mode.py ............. 2066
84.64 pylab_examples example code: demo_tight_layout.py ...................... 2067
84.65 pylab_examples example code: dolphin.py ................................ 2077
84.66 pylab_examples example code: ellipse_collection.py .................... 2080
84.67 pylab_examples example code: ellipse_demo.py ............................ 2081
84.68 pylab_examples example code: ellipse_rotated.py ....................... 2082
<table>
<thead>
<tr>
<th>Example Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal_aspect_ratio.py</td>
<td>2083</td>
</tr>
<tr>
<td>errorbar_limits.py</td>
<td>2084</td>
</tr>
<tr>
<td>errorbar_subsample.py</td>
<td>2086</td>
</tr>
<tr>
<td>eventcollection_demo.py</td>
<td>2088</td>
</tr>
<tr>
<td>eventplot_demo.py</td>
<td>2090</td>
</tr>
<tr>
<td>fancyarrow_demo.py</td>
<td>2092</td>
</tr>
<tr>
<td>fancybox_demo.py</td>
<td>2094</td>
</tr>
<tr>
<td>fancybox_demo2.py</td>
<td>2098</td>
</tr>
<tr>
<td>fancytextbox_demo.py</td>
<td>2099</td>
</tr>
<tr>
<td>figimage_demo.py</td>
<td>2100</td>
</tr>
<tr>
<td>figlegend_demo.py</td>
<td>2101</td>
</tr>
<tr>
<td>figure_title.py</td>
<td>2102</td>
</tr>
<tr>
<td>fill_between_demo.py</td>
<td>2104</td>
</tr>
<tr>
<td>fill_betweenx_demo.py</td>
<td>2108</td>
</tr>
<tr>
<td>fill_spiral.py</td>
<td>2111</td>
</tr>
<tr>
<td>finance_demo.py</td>
<td>2112</td>
</tr>
<tr>
<td>finance_work2.py</td>
<td>2114</td>
</tr>
<tr>
<td>findobj_demo.py</td>
<td>2119</td>
</tr>
<tr>
<td>font_table_ttf.py</td>
<td>2120</td>
</tr>
<tr>
<td>fonts_demo.py</td>
<td>2122</td>
</tr>
<tr>
<td>fonts_demo_kw.py</td>
<td>2125</td>
</tr>
<tr>
<td>ganged_plots.py</td>
<td>2127</td>
</tr>
<tr>
<td>geo_demo.py</td>
<td>2129</td>
</tr>
<tr>
<td>ginput_demo.py</td>
<td>2130</td>
</tr>
<tr>
<td>ginput_manual_clabel.py</td>
<td>2130</td>
</tr>
<tr>
<td>gradient_manual_clabel.py</td>
<td>2132</td>
</tr>
<tr>
<td>griddata_demo.py</td>
<td>2134</td>
</tr>
<tr>
<td>hatch_demo.py</td>
<td>2135</td>
</tr>
<tr>
<td>hexbin_demo.py</td>
<td>2136</td>
</tr>
<tr>
<td>hexbin_demo2.py</td>
<td>2138</td>
</tr>
<tr>
<td>hist2d_demo.py</td>
<td>2140</td>
</tr>
<tr>
<td>hist2d_log_demo.py</td>
<td>2141</td>
</tr>
<tr>
<td>hist_colormapped.py</td>
<td>2142</td>
</tr>
<tr>
<td>histogram_percent_demo.py</td>
<td>2143</td>
</tr>
<tr>
<td>hyperlinks.py</td>
<td>2144</td>
</tr>
<tr>
<td>image_clip_path.py</td>
<td>2145</td>
</tr>
<tr>
<td>image_demo.py</td>
<td>2146</td>
</tr>
<tr>
<td>image_demo2.py</td>
<td>2147</td>
</tr>
<tr>
<td>image_interp.py</td>
<td>2149</td>
</tr>
<tr>
<td>image_masked.py</td>
<td>2153</td>
</tr>
<tr>
<td>image_nonuniform.py</td>
<td>2155</td>
</tr>
<tr>
<td>image_origin.py</td>
<td>2157</td>
</tr>
<tr>
<td>image_slices_viewer.py</td>
<td>2158</td>
</tr>
<tr>
<td>interp_demo.py</td>
<td>2160</td>
</tr>
<tr>
<td>invert_axes.py</td>
<td>2161</td>
</tr>
<tr>
<td>layer_images.py</td>
<td>2162</td>
</tr>
<tr>
<td>legend_demo2.py</td>
<td>2164</td>
</tr>
<tr>
<td>legend_demo3.py</td>
<td>2165</td>
</tr>
</tbody>
</table>
84.165    pylab_examples example code: scatter_demo2.py
84.166    pylab_examples example code: scatter_hist.py
84.167    pylab_examples example code: scatter_masked.py
84.168    pylab_examples example code: scatter_profile.py
84.169    pylab_examples example code: scatter_star_poly.py
84.170    pylab_examples example code: scatter_symbol.py
84.171    pylab_examples example code: set_and_get.py
84.172    pylab_examples example code: shading_example.py
84.173    pylab_examples example code: shared_axis_across_figures.py
84.174    pylab_examples example code: shared_axis_demo.py
84.175    pylab_examples example code: simple_plot.py
84.176    pylab_examples example code: simple_plot_fps.py
84.177    pylab_examples example code: specgram_demo.py
84.178    pylab_examples example code: spectrum_demo.py
84.179    pylab_examples example code: spine-placement_demo.py
84.180    pylab_examples example code: spy_demos.py
84.181    pylab_examples example code: stackplot_demo.py
84.182    pylab_examples example code: stackplot_demo2.py
84.183    pylab_examples example code: step_demo.py
84.184    pylab_examples example code: six_fonts_demo.py
84.185    pylab_examples example code: stock_demo.py
84.186    pylab_examples example code: subplot_demo.py
84.187    pylab_examples example code: subplot_toolbar.py
84.188    pylab_examples example code: subplots_adjust.py
84.189    pylab_examples example code: subplots_adjust.py
84.190    pylab_examples example code: symlog_demo.py
84.191    pylab_examples example code: table_demo.py
84.192    pylab_examples example code: tex_demo.py
84.193    pylab_examples example code: tex_unicode_demo.py
84.194    pylab_examples example code: text_handles.py
84.195    pylab_examples example code: text_rotation.py
84.196    pylab_examples example code: text_rotation_relative_to_line.py
84.197    pylab_examples example code: titles_demo.py
84.198    pylab_examples example code: to_numeric.py
84.199    pylab_examples example code: toggle_images.py
84.200    pylab_examples example code: transoffset.py
84.201    pylab_examples example code: tricontour_demo.py
84.202    pylab_examples example code: tricontour_smooth_delaunay.py
84.203    pylab_examples example code: tricontour_smooth_user.py
84.204    pylab_examples example code: tricontour_vs_griddata.py
84.205    pylab_examples example code: trigradient_demo.py
84.206    pylab_examples example code: triinterpl.py
84.207    pylab_examples example code: usetex_base.py
84.208    pylab_examples example code: usetex_quad.py
84.209    pylab_examples example code: usetex_quad.py
84.210    pylab_examples example code: usetex_quad.py
84.211    pylab_examples example code: usetex_quad.py
84.212    pylab_examples example code: usetex_quad.py
84.213    pylab_examples example code: usetex_quad.py
84.214    pylab_examples example code: usetex_quad.py
84.215    pylab_examples example code: usetex_quad.py
84.216    pylab_examples example code: usetex_quad.py
84.217    pylab_examples example code: usetex_quad.py
84.218    pylab_examples example code: usetex_quad.py
84.219    pylab_examples example code: usetex_quad.py
84.220    pylab_examples example code: usetex_quad.py
84.221    pylab_examples example code: usetex_quad.py
84.222    pylab_examples example code: usetex_quad.py
84.223    pylab_examples example code: usetex_quad.py
84.224    pylab_examples example code: usetex_quad.py
84.225    pylab_examples example code: usetex_quad.py
84.226    pylab_examples example code: usetex_quad.py
84.227    pylab_examples example code: usetex_quad.py
84.228    pylab_exa...
## $\text{85 shapes_and_collections Examples}$

- **85.1** shapes_and_collections example code: artist_reference.py ........................................... 2359
- **85.2** shapes_and_collections example code: path_patch_demo.py .......................................... 2362
- **85.3** shapes_and_collections example code: scatter_demo.py ............................................. 2363

## $\text{86 showcase Examples}$

- **86.1** showcase example code: integral_demo.py ............................................................... 2365
- **86.2** showcase example code: xkcd.py .................................................................................. 2367

## $\text{87 specialty_plots Examples}$

- **87.1** specialty_plots example code: hinton_demo.py .......................................................... 2371

## $\text{88 statistics Examples}$

- **88.1** statistics example code: boxplot_demo.py ................................................................. 2373
- **88.2** statistics example code: bxp_demo.py .......................................................................... 2376
- **88.3** statistics example code: errorbar_demo.py ................................................................. 2379
- **88.4** statistics example code: errorbar_demo_features.py ...................................................... 2380
- **88.5** statistics example code: errorbar_limits.py ............................................................... 2382
- **88.6** statistics example code: histogram_demo_cumulative.py ............................................. 2384
- **88.7** statistics example code: histogram_demo_features.py .................................................. 2385
- **88.8** statistics example code: histogram_demo_histtypes.py .................................................. 2386
- **88.9** statistics example code: histogram_demo_multihist.py .................................................. 2388
- **88.10** statistics example code: violinplot_demo.py ............................................................. 2390

## $\text{89 style_sheets Examples}$

- **89.1** style_sheets example code: plot_bmh.py ...................................................................... 2393
- **89.2** style_sheets example code: plot_dark_background.py .................................................. 2394
- **89.3** style_sheets example code: plot_fivethirtyeight.py ....................................................... 2395
- **89.4** style_sheets example code: plot_ggplot.py .................................................................. 2396
- **89.5** style_sheets example code: plot_grayscale.py ............................................................ 2398

## $\text{90 subplots_axes_and_figures Examples}$

- **90.1** subplots_axes_and_figures example code: fahrenheit_celsius_scales.py ....................... 2401
- **90.2** subplots_axes_and_figures example code: subplot_demo.py ........................................... 2403

## $\text{91 tests Examples}$

- **91.1** tests example code: backend_driver.py ...................................................................... 2405

## $\text{92 text_labels_and_annotations Examples}$

- **92.1** text_labels_and_annotations example code: rainbow_text.py ..................................... 2417
- **92.2** text_labels_and_annotations example code: text_demo_fontdict.py ............................ 2419
- **92.3** text_labels_and_annotations example code: unicode_demo.py ...................................... 2420
93 ticks_and_spines Examples
  93.1 ticks_and_spines example code: spines_demo.py .......................... 2423
  93.2 ticks_and_spines example code: spines_demo_bounds.py ............... 2425
  93.3 ticks_and_spines example code: spines_demo_dropped.py .............. 2426
  93.4 ticks_and_spines example code: ticklabels_demo_rotation.py ....... 2427

94 units Examples
  94.1 units example code: annotate_with_units.py ............................. 2429
  94.2 units example code: artist_tests.py ................................. 2430
  94.3 units example code: bar_demo2.py ................................... 2432
  94.4 units example code: bar_unit_demo.py .................................. 2433
  94.5 units example code: basic_units.py .................................... 2434
  94.6 units example code: ellipse_with_units.py ............................ 2442
  94.7 units example code: evans_test.py ..................................... 2445
  94.8 units example code: radian_demo.py .................................. 2447
  94.9 units example code: units_sample.py .................................. 2448
  94.10 units example code: units_scatter.py ................................ 2450

95 user_interfaces Examples
  95.1 user_interfaces example code: embedding_in_gtk.py ...................... 2453
  95.2 user_interfaces example code: embedding_in_gtk2.py ................... 2454
  95.3 user_interfaces example code: embedding_in_gtk3.py ................... 2455
  95.4 user_interfaces example code: embedding_in_gtk3_panzoom.py ......... 2455
  95.5 user_interfaces example code: embedding_in_qt4.py .................... 2456
  95.6 user_interfaces example code: embedding_in_qt4_toolbar.py .......... 2459
  95.7 user_interfaces example code: embedding_in_tk.py .................... 2461
  95.8 user_interfaces example code: embedding_in_tk2.py ................... 2462
  95.9 user_interfaces example code: embedding_in_wx2.py ................... 2463
  95.10 user_interfaces example code: embedding_in_wx3.py .................... 2465
  95.11 user_interfaces example code: embedding_in_wx4.py ................... 2468
  95.12 user_interfaces example code: embedding_in_wx5.py ................... 2470
  95.13 user_interfaces example code: embedding_webagg.py .................. 2471
  95.14 user_interfaces example code: fourier_demo_wx.py ................... 2476
  95.15 user_interfaces example code: gtk_spreadsheet.py .................... 2480
  95.16 user_interfaces example code: histogram_demo_canvasagg.py ........ 2482
  95.17 user_interfaces example code: interactive.py ....................... 2484
  95.18 user_interfaces example code: interactive2.py ....................... 2488
  95.19 user_interfaces example code: lineprops_dialog_gtk.py .............. 2496
  95.20 user_interfaces example code: mathtext_wx.py ....................... 2496
  95.21 user_interfaces example code: mpl_with_glade.py .................... 2499
  95.22 user_interfaces example code: mpl_with_glade_316.py ............... 2501
  95.23 user_interfaces example code: pylab_with_gtk.py .................... 2502
  95.24 user_interfaces example code: rec_edit_gtk_custom.py ............... 2503
  95.25 user_interfaces example code: rec_edit_gtk_simple.py ............... 2504
  95.26 user_interfaces example code: svg_histogram.py ..................... 2504
  95.27 user_interfaces example code: svg_tooltip.py ....................... 2508
  95.28 user_interfaces example code: wxcursor_demo.py ..................... 2510
96 widgets Examples
   96.1 widgets example code: buttons.py ........................................ 2513
   96.2 widgets example code: check_buttons.py .................................. 2514
   96.3 widgets example code: cursor.py ........................................... 2514
   96.4 widgets example code: lasso_selector_demo.py ............................ 2515
   96.5 widgets example code: menu.py ............................................. 2517
   96.6 widgets example code: multicursor.py ...................................... 2520
   96.7 widgets example code: radio_buttons.py .................................... 2521
   96.8 widgets example code: rectangle_selector.py ............................... 2522
   96.9 widgets example code: slider_demo.py ...................................... 2523
   96.10 widgets example code: span_selector.py ................................... 2524

X Glossary ................................................................. 2527

Bibliography ............................................................... 2531

Python Module Index ..................................................... 2533

Index .............................................................................. 2535
Part I

User’s Guide
INTRODUCTION

matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB® graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way. Although matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays.

matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn’t need to instantiate objects, call methods, set properties, and so on; it should just work.

For years, I used to use MATLAB exclusively for data analysis and visualization. MATLAB excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed an EEG analysis application in MATLAB. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of MATLAB as a programming language, and decided to start over in Python. Python more than makes up for all of MATLAB’s deficiencies as a programming language, but I was having difficulty finding a 2D plotting package (for 3D VTK more than exceeds all of my needs).

When I went searching for a Python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

Finding no package that suited me just right, I did what any self-respecting Python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate MATLAB’s plotting capabilities because that is something MATLAB does very well. This had the added advantage that many people have a lot of MATLAB experience, and thus they can quickly get up to steam plotting in python. From a developer’s perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

The matplotlib code is conceptually divided into three parts: the pylab interface is the set of functions provided by matplotlib.pylab which allow the user to create plots with code quite similar to MATLAB.

---

1 MATLAB is a registered trademark of The MathWorks, Inc.
figure generating code (*Pyplot tutorial*). The *matplotlib frontend* or *matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on (*Artist tutorial*). This is an abstract interface that knows nothing about output. The *backends* are device-dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device (*What is a backend?*).

Example backends: PS creates PostScript® hardcopy, SVG creates Scalable Vector Graphics hardcopy, Agg creates PNG output using the high quality Anti-Grain Geometry library that ships with matplotlib, GTK embeds matplotlib in a Gtk+ application, GTKAgg uses the Anti-Grain renderer to create a figure and embed it in a Gtk+ application, and so on for PDF, WxWidgets, Tkinter, etc.

matplotlib is used by many people in many different contexts. Some people want to automatically generate PostScript files to send to a printer or publishers. Others deploy matplotlib on a web application server to generate PNG output for inclusion in dynamically-generated web pages. Some use matplotlib interactively from the Python shell in Tkinter on Windows™. My primary use is to embed matplotlib in a Gtk+ EEG application that runs on Windows, Linux and Macintosh OS X.
2.1 Installing

There are many different ways to install matplotlib, and the best way depends on what operating system you are using, what you already have installed, and how you want to use it. To avoid wading through all the details (and potential complications) on this page, there are several convenient options.

2.1.1 Installing pre-built packages

Most platforms : scientific Python distributions

The first option is to use one of the pre-packaged python distributions that already provide matplotlib built-in. The Continuum.io Python distribution (Anaconda or miniconda) and the Enthought distribution (Canopy) are both excellent choices that “just work” out of the box for Windows, OSX and common Linux platforms. Both of these distributions include matplotlib and lots of other useful tools. Another excellent alternative for Windows users is Python (x, y).

Linux : using your package manager

If you are on Linux, you might prefer to use your package manager. matplotlib is packaged for almost every major Linux distribution.

- Debian / Ubuntu: `sudo apt-get install python-matplotlib`
- Fedora / Redhat: `sudo yum install python-matplotlib`

Mac OSX : using pip

If you are on Mac OSX you can probably install matplotlib binaries using the standard Python installation program pip. See Installing OSX binary wheels.
Windows

If you don’t already have Python installed, we recommend using one of the scipy-stack compatible Python distributions such as Python(x,y), Enthought Canopy, or Continuum Anaconda, which have matplotlib and many of its dependencies, plus other useful packages, preinstalled.

For standard Python installations you will also need to install compatible versions of setuptools, numpy, python-dateutil, pytz, pyparsing and six in addition to matplotlib.

In case Python is not installed for all users (not the default), the Microsoft Visual C++ 2008 (64 bit or 32 bit for Python 2.6 to 3.2) or Microsoft Visual C++ 2010 (64 bit or 32 bit for Python 3.3 and 3.4) redistributable packages need to be installed.

Matplotlib depends on Pillow for reading and saving JPEG, BMP, and TIFF image files. Matplotlib requires MiKTeX and GhostScript for rendering text with LaTeX. FFmpeg, avconv, mencoder, or ImageMagick are required for the animation module.

The following backends should work out of the box: agg, tkagg, ps, pdf and svg. For other backends you may need to install pycairo, PyQt4, PyQt5, PySide, wxPython, PyGTK, Tornado, or GhostScript.

TkAgg is probably the best backend for interactive use from the standard Python shell or IPython. It is enabled as the default backend for the official binaries. GTK3 is not supported on Windows.

The Windows installers (*.exe) and wheels (*.whl) on the download page do not contain test data or example code. If you want to try the many demos that come in the matplotlib source distribution, download the *.tar.gz file and look in the examples subdirectory. To run the test suite, copy the libmatplotlibtests and libmpl_toolkitstests directories from the source distribution to sys.prefixLibsite-packagesmatplotlib and sys.prefixLibsite-packagesmpl_toolkits respectively, and install nose, mock, Pillow, MiKTeX, GhostScript, ffmpeg, avconv, mencoder, ImageMagick, and Inkscape.

2.1.2 Installing from source

If you are interested in contributing to matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build matplotlib from source. Grab the latest tar.gz release file from the download page, or if you want to develop matplotlib or just need the latest bugfixed version, grab the latest git version Source install from git.

Once you have satisfied the requirements detailed below (mainly python, numpy, libpng and freetype), you can build matplotlib:

cd matplotlib
python setup.py build
python setup.py install

We provide a setup.cfg file that goes with setup.py which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging matplotlib.

If you have installed prerequisites to nonstandard places and need to inform matplotlib where they are, edit setupext.py and add the base dirs to the basedir dictionary entry for your sys.platform. e.g., if
the header to some required library is in /some/path/include/someheader.h, put /some/path in the basedir list for your platform.

**Build requirements**

These are external packages which you will need to install before installing matplotlib. If you are building on OSX, see *Building on OSX*. If you are building on Windows, see *Building on Windows*. If you are installing dependencies with a package manager on Linux, you may need to install the development packages (look for a “-dev” postfix) in addition to the libraries themselves.

**Required Dependencies**

- **python 2.6, 2.7, 3.3 or 3.4**  Download python.
- **numpy 1.6 (or later)**  array support for python (download numpy)
- **dateutil 1.1 or later**  Provides extensions to python datetime handling. If using pip, easy_install or installing from source, the installer will attempt to download and install python_dateutil from PyPI. Note that python_dateutil also depends on six. pip and other package managers should handle installing that secondary dependency automatically.
- **pyparsing**  Required for matplotlib’s mathtext math rendering support. If using pip, easy_install or installing from source, the installer will attempt to download and install pyparsing from PyPI.
- **six 1.4 or later**  Python 2/3 compatibility library. This is also a dependency of dateutil.
- **libpng 1.2 (or later)**  library for loading and saving PNG files (download). libpng requires zlib.
- **pytz**  Used to manipulate time-zone aware datetimes.
- **freetype 2.3 or later**  library for reading true type font files.

**Optional GUI framework**

These are optional packages which you may want to install to use matplotlib with a user interface toolkit. See *What is a backend?* for more details on the optional matplotlib backends and the capabilities they provide.

- **tk 8.3 or later**  The TCL/Tk widgets library used by the TkAgg backend
- **pyqt 4.0 or later**  The Qt4 widgets library python wrappers for the Qt4Agg backend
- **pygtk 2.4 or later**  The python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend
- **wxpython 2.8 or later**  The python wrappers for the wx widgets library for use with the WX or WXAgg backend

2.1. Installing
Optional external programs

ffmpeg/avconv or mencoder  Required for the animation module to be save out put to movie formats.
ImageMagick  Required for the animation module to be able to save to animated gif.

Optional dependencies

Pillow  If Pillow is installed, matplotlib can read and write a larger selection of image file formats.

Required libraries that ship with matplotlib

agg 2.4  The antigrain C++ rendering engine. matplotlib links against the agg template source statically, so it will not affect anything on your system outside of matplotlib.
PyCXX 6.2.4  A library for writing Python extensions in C++.
qhull 2012.1  A library for computing convex hulls. Used for computing triangulation and meshes.
ttconv  truetype font utility

Building on Linux

It is easiest to use your system package manager to install the dependencies.
If you are on Debian/Ubuntu, you can get all the dependencies required to build matplotlib with:
sudo apt-get build-dep python-matplotlib

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing yum-builddep and then running:
su -c "yum-builddep python-matplotlib"

This does not build matplotlib, but it does get the install the build dependencies, which will make building from source easier.

Building on OSX

The build situation on OSX is complicated by the various places one can get the libpng and freetype requirements (darwinports, fink, /usr/X11R6) and the different architectures (e.g., x86, ppc, universal) and the different OSX version (e.g., 10.4 and 10.5). We recommend that you build the way we do for the OSX release: get the source from the tarball or the git repository and follow the instruction in README.osx.
Building on Windows

The Python shipped from http://www.python.org is compiled with Visual Studio 2008 for versions before 3.3 and Visual Studio 2010 for 3.3 and later. Python extensions are recommended to be compiled with the same compiler. The .NET Framework 4.0 is required for MSBuild (you’ll likely have the requisite Framework with Visual Studio). In addition to Visual Studio CMake is required for building libpng.

Since there is no canonical Windows package manager the build methods for freetype, zlib, libpng, tcl, & tk source code are documented as a build script at matplotlib-winbuild.

2.2 Customizing matplotlib

2.2.1 The matplotlibrc file

matplotlib uses matplotlibrc configuration files to customize all kinds of properties, which we call rc settings or rc parameters. You can control the defaults of almost every property in matplotlib: figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties and so on. matplotlib looks for matplotlibrc in three locations, in the following order:

1. matplotlibrc in the current working directory, usually used for specific customizations that you do not want to apply elsewhere.

2. It next looks in a user-specific place, depending on your platform:

   - On Linux, it looks in .config/matplotlib/matplotlibrc (or $XDG_CONFIG_HOME/matplotlib/matplotlibrc) if you’ve customized your environment.

   - On other platforms, it looks in .matplotlib/matplotlibrc.

   See .matplotlib directory location.

3. INSTALL/matplotlib/mpl-data/matplotlibrc, where INSTALL is something like /usr/lib/python2.5/site-packages on Linux, and maybe C:\Python25\Lib\site-packages on Windows. Every time you install matplotlib, this file will be overwritten, so if you want your customizations to be saved, please move this file to your user-specific matplotlib directory.

To display where the currently active matplotlibrc file was loaded from, one can do the following:

```python
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'/home/foo/.config/matplotlib/matplotlibrc'
```

See below for a sample matplotlibrc file.

2.2.2 Dynamic rc settings

You can also dynamically change the default rc settings in a python script or interactively from the python shell. All of the rc settings are stored in a dictionary-like variable called matplotlib.rcParams, which is
global to the matplotlib package. rcParams can be modified directly, for example:

```python
import matplotlib as mpl
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.color'] = 'r'
```

Matplotlib also provides a couple of convenience functions for modifying rc settings. The `matplotlib.rc()` command can be used to modify multiple settings in a single group at once, using keyword arguments:

```python
import matplotlib as mpl
mpl.rc('lines', linewidth=2, color='r')
```

The `matplotlib.rcdefaults()` command will restore the standard matplotlib default settings.

There is some degree of validation when setting the values of rcParams, see `matplotlib.rcsetup` for details.

### A sample matplotlibrc file

```plaintext
### MATPLOTLIBRC FORMAT

# This is a sample matplotlib configuration file - you can find a copy
# of it on your system in
# site-packages/matplotlib mpl-data/matplotlibrc. If you edit it
# there, please note that it will be overwritten in your next install.
# If you want to keep a permanent local copy that will not be
# overwritten, place it in the following location:
# unix/linux:
#   $HOME/.config/matplotlib/matplotlibrc or
#   $XDG_CONFIG_HOME/matplotlib/matplotlibrc (if $XDG_CONFIG_HOME is set)
# other platforms:
#   $HOME/.matplotlib/matplotlibrc
#
# See http://matplotlib.org/users/customizing.html#the-matplotlibrc-file for
# more details on the paths which are checked for the configuration file.
#
# This file is best viewed in a editor which supports python mode
# syntax highlighting. Blank lines, or lines starting with a comment
# symbol, are ignored, as are trailing comments. Other lines must
# have the format
#   key : val # optional comment
#
# Colors: for the color values below, you can either use - a
# matplotlib color string, such as r, k, or b - an rgb tuple, such as
# (1.0, 0.5, 0.0) - a hex string, such as #ff00ff or ff00ff - a scalar
# grayscale intensity such as 0.75 - a legal html color name, e.g., red,
# blue, darkslategray

### CONFIGURATION BEGINS HERE

# The default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo
# CocoaAgg MacOSX Qt4Agg Qt5Agg TkAgg WX WXAgg Agg Cairo GDK PS PDF SVG
```
# Template.
# You can also deploy your own backend outside of matplotlib by
# referring to the module name (which must be in the PYTHONPATH) as
# 'module://my_backend'.
backend : qt4agg

# If you are using the Qt4Agg backend, you can choose here
# to use the PyQt4 bindings or the newer PySide bindings to
# the underlying Qt4 toolkit.
#backend.qt4 : PyQt4 # PyQt4 | PySide

# Note that this can be overridden by the environment variable
# QT_API used by Enthought Tool Suite (ETS); valid values are
# "pyqt" and "pyside". The "pyqt" setting has the side effect of
# forcing the use of Version 2 API for QString and QVariant.

# The port to use for the web server in the WebAgg backend.
# webagg.port : 8888

# If webagg.port is unavailable, a number of other random ports will
# be tried until one that is available is found.
# webagg.port_retries : 50

# When True, open the webbrowser to the plot that is shown
# webagg.open_in_browser : True

# When True, the figures rendered in the nbagg backend are created with
# a transparent background.
# nbagg.transparent : True

# if you are running pyplot inside a GUI and your backend choice
# conflicts, we will automatically try to find a compatible one for
# you if backend_fallback is True
#backend_fallback: True

#interactive : False
#toolbar : toolbar2 # None | toolbar2 ("classic" is deprecated)
#timezone : UTC # a pytz timezone string, e.g., US/Central or Europe/Paris

# Where your matplotlib data lives if you installed to a non-default
# location. This is where the matplotlib fonts, bitmaps, etc reside
#datapath : /home/jdhunter/mpldata

### LINES
# See http://matplotlib.org/api/artist_api.html#module-matplotlib.lines for more
# information on line properties.
#lines.linewidth : 1.0 # line width in points
#lines linestyle : - # solid line
#lines color : blue # has no affect on plot(); see axes.color_cycle
#lines marker : None # the default marker
#lines markeredgewidth : 0.5 # the line width around the marker symbol
#lines markersize : 6 # markersize, in points
### PATCHES

# Patches are graphical objects that fill 2D space, like polygons or circles. See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.patches
# information on patch properties
patch.linewidth : 1.0    # edge width in points
patch.facecolor : blue
patch.edgecolor : black
patch.antialiased : True    # render patches in antialiased (no jaggies)

### FONT

# font properties used by text.Text. See
# http://matplotlib.org/api/font_manager_api.html for more
# information on font properties. The 6 font properties used for font
# matching are given below with their default values.
#
# The font.family property has five values: 'serif' (e.g., Times),
# 'sans-serif' (e.g., Helvetica), 'cursive' (e.g., Zapf-Chancery),
# 'fantasy' (e.g., Western), and 'monospace' (e.g., Courier). Each of
# these font families has a default list of font names in decreasing
# order of priority associated with them. When text.usetex is False,
# font.family may also be one or more concrete font names.
#
# The font.style property has three values: normal (or roman), italic
# or oblique. The oblique style will be used for italic, if it is not
# present.
#
# The font.variant property has two values: normal or small-caps. For
# TrueType fonts, which are scalable fonts, small-caps is equivalent
# to using a font size of 'smaller', or about 83% of the current font
# size.
#
# The font.weight property has effectively 13 values: normal, bold,
# bolder, lighter, 100, 200, 300, ..., 900. Normal is the same as
# 400, and bold is 700. bolder and lighter are relative values with
# respect to the current weight.
#
# The font.stretch property has 11 values: ultra-condensed,
# extra-condensed, condensed, semi-condensed, normal, semi-expanded,
# expanded, extra-expanded, ultra-expanded, wider, and narrower. This
# property is not currently implemented.
#
# The font.size property is the default font size for text, given in pts.
# 12pt is the standard value.
#
# font.family : sans-serif
Matplotlib, Release 1.4.3

# font.style : normal
# font.variant : normal
# font.weight : medium
# font.stretch : normal
# note that font.size controls default text sizes. To configure
# special text sizes tick labels, axes, labels, title, etc, see the rc
# settings for axes and ticks. Special text sizes can be defined
# relative to font.size, using the following values: xx-small, x-small,
# small, medium, large, x-large, xx-large, larger, or smaller
#font.size : 12.0
#font.serif : Bitstream Vera Serif, New Century Schoolbook, Century Schoolbook L, Utopia, ITC Bookman
#font.sans-serif : Bitstream Vera Sans, Lucida Grande, Verdana, Geneva, Lucid, Arial, Helvetica, Avant Garde
#font.cursive : Apple Chancery, Textile, Zapf Chancery, Sand, cursive
#font.fantasy : Comic Sans MS, Chicago, Charcoal, Impact, Western, fantasy
#font.monospace : Bitstream Vera Sans Mono, Andale Mono, Nimbus Mono L, Courier New, Courier, Fixedsys

### TEXT
# text properties used by text.Text. See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.text for more
# information on text properties

text.color : black

### LaTeX customizations. See http://www.scipy.org/Wiki/Cookbook/Matplotlib/UsingTex
# text.usetex : False # use latex for all text handling. The following fonts
# are supported through the usual rc parameter settings:
# new century schoolbook, bookman, times, palatino,
# zapf chancery, charter, serif, sans-serif, helvetica,
# avant garde, courier, monospace, computer modern roman,
# computer modern sans serif, computer modern typewriter
# If another font is desired which can loaded using the
# \LaTeX \usepackage command, please inquire at the
# matplotlib mailing list

text.latex.unicode : False # use "ucs" and "inputenc" \LaTeX packages for handling
# unicode strings.

text.latex.preamble : # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX FAILURES
# AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR HELP
# IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
# preamble is a comma separated list of \LaTeX statements
# that are included in the \LaTeX document preamble.
# An example:
# text.latex.preamble : \usepackage{bm},\usepackage{euler}
# The following packages are always loaded with usetex, so
# beware of package collisions: color, geometry, graphicx,
# typelcm, textcomp. Adobe Postscript (PSSNFS) font packages
# may also be loaded, depending on your font settings

text.dvipnghack : None # some versions of dvipng don't handle alpha
# channel properly. Use True to correct
# and flush ~/.matplotlib/tex.cache
# before testing and False to force
# correction off. None will try and
# guess based on your dvipng version

2.2. Customizing matplotlib 13
#text.hinting : auto
# May be one of the following:
# 'none': Perform no hinting
# 'auto': Use freetype's autohinter
# 'native': Use the hinting information in the
# font file, if available, and if your
# freetype library supports it
# 'either': Use the native hinting information,
# or the autohinter if none is available.
# For backward compatibility, this value may also be
# True === 'auto' or False === 'none'.
#text.hinting_factor : 8
# Specifies the amount of softness for hinting in the
# horizontal direction. A value of 1 will hint to full
# pixels. A value of 2 will hint to half pixels etc.
#text.antialiased : True
# If True (default), the text will be antialiased.
# This only affects the Agg backend.

# The following settings allow you to select the fonts in math mode.
# They map from a TeX font name to a fontconfig font pattern.
# These settings are only used if mathtext.fontset is 'custom'.
# Note that this "custom" mode is unsupported and may go away in the
# future.
#mathtext.cal : cursive
#mathtext.rm : serif
#mathtext.tt : monospace
#mathtext.it : serif:italic
#mathtext.bf : serif:bold
#mathtext.sf : sans
#mathtext.fontset : cm # Should be 'cm' (Computer Modern), 'stix',
# 'stixsans' or 'custom'
#mathtext.fallback_to_cm : True # When True, use symbols from the Computer Modern
# fonts when a symbol can not be found in one of
# the custom math fonts.
#mathtext.default : it # The default font to use for math.
# Can be any of the LaTeX font names, including
# the special name "regular" for the same font
# used in regular text.

### AXES
# default face and edge color, default tick sizes,
# default fontsize for ticklabels, and so on. See
# http://matplotlib.org/api/axes_api.html#module-matplotlib.axes
#axes.hold  : True  # whether to clear the axes by default on
#axes.facecolor : white  # axes background color
#axes.edgecolor : black  # axes edge color
#axes.linewidth : 1.0  # edge linewidth
#axes.grid : False  # display grid or not
#axes.titlesize : large  # fontsize of the axes title
#axes.labelsize : medium  # fontsize of the x and y labels
#axes.labelweight : normal  # weight of the x and y labels
#axes.labelcolor : black
#axes.axisbelow : False  # whether axis gridlines and ticks are below
# the axes elements (lines, text, etc)

#axes.formatter.limits : -7, 7 # use scientific notation if log10
    # of the axis range is smaller than the
    # first or larger than the second
#axes.formatter.use_locale : False # When True, format tick labels
    # according to the user's locale.
    # For example, use ',' as a decimal
    # separator in the fr_FR locale.
#axes.formatter.use_mathtext : False # When True, use mathtext for scientific
    # notation.
#axes.formatter.useoffset : True # If True, the tick label formatter
    # will default to labeling ticks relative
    # to an offset when the data range is very
    # small compared to the minimum absolute
    # value of the data.

#axes.unicode_minus : True # use unicode for the minus symbol
    # rather than hyphen. See
    # http://en.wikipedia.org/wiki/Plus_and_minus_signs#Character_codes
#axes.color_cycle : b, g, r, c, m, y, k # color cycle for plot lines
    # as list of string colorspecs:
    # single letter, long name, or
    # web-style hex
#axes.xmargin : 0 # x margin. See
#axes.ymargin : 0 # y margin See `axes.Axes.margins`

#polaraxes.grid : True # display grid on polar axes
#axes3d.grid : True # display grid on 3d axes

### TICKS
# see http://matplotlib.org/api/axis_api.html#matplotlib.axis.Tick
#xtick.major.size : 4 # major tick size in points
#xtick.minor.size : 2 # minor tick size in points
#xtick.major.width : 0.5 # major tick width in points
#xtick.minor.width : 0.5 # minor tick width in points
#xtick.major.pad : 4 # distance to major tick label in points
#xtick.minor.pad : 4 # distance to the minor tick label in points
#xtick.color : k # color of the tick labels
#xtick.labelsize : medium # fontsize of the tick labels
#xtick.direction : in # direction: in, out, or inout

#ytick.major.size : 4 # major tick size in points
#ytick.minor.size : 2 # minor tick size in points
#ytick.major.width : 0.5 # major tick width in points
#ytick.minor.width : 0.5 # minor tick width in points
#ytick.major.pad : 4 # distance to major tick label in points
#ytick.minor.pad : 4 # distance to the minor tick label in points
#ytick.color : k # color of the tick labels
#ytick.labelsize : medium # fontsize of the tick labels
#ytick.direction : in # direction: in, out, or inout

2.2. Customizing matplotlib
### GRIDS

- **grid.color**: black  # grid color
- **grid.linestyle**: :  # dotted
- **grid.linewidth**: 0.5  # in points
- **grid.alpha**: 1.0  # transparency, between 0.0 and 1.0

### Legend

- **legend.fancybox**: False  # if True, use a rounded box for the legend, else a rectangle
- **legend.isaxes**: True
- **legend.numpoints**: 2  # the number of points in the legend line
- **legend.fontsize**: large
- **legend.borderpad**: 0.5  # border whitespace in fontsize units
- **legend.markerscale**: 1.0  # the relative size of legend markers vs. original
- **legend.labelspacing**: 0.5  # the vertical space between the legend entries in fraction of fontsize
- **legend.handlelength**: 2.0  # the length of the legend handle in fraction of fontsize
- **legend.handleheight**: 0.7  # the height of the legend handle in fraction of fontsize
- **legend.handletextpad**: 0.5  # the space between the legend line and legend text in fraction of fontsize
- **legend.borderaxespad**: 0.5  # the border between the axes and legend edge in fraction of fontsize
- **legend.shadow**: False
- **legend.frameon**: True  # whether or not to draw a frame around legend
- **legend.framealpha**: None  # opacity of of legend frame
- **legend.scatterpoints**: 3  # number of scatter points

### FIGURE

- **figure.figsize**: 8, 6  # figure size in inches
- **figure.dpi**: 80  # figure dots per inch
- **figure.facecolor**: 0.75  # figure facecolor; 0.75 is scalar gray
- **figure.edgecolor**: white  # figure edgecolor
- **figure.autolayout**: False  # When True, automatically adjust subplot parameters to make the plot fit the figure
- **figure.max_open_warning**: 20  # The maximum number of figures to open through the pyplot interface before emitting a warning. If less than one this feature is disabled.

# The figure subplot parameters. All dimensions are a fraction of the figure

- **figure.subplot.left**: 0.125  # the left side of the subplots of the figure
- **figure.subplot.right**: 0.9  # the right side of the subplots of the figure
- **figure.subplot.bottom**: 0.1  # the bottom of the subplots of the figure
- **figure.subplot.top**: 0.9  # the top of the subplots of the figure
- **figure.subplot.wspace**: 0.2  # the amount of width reserved for blank space between subplots
- **figure.subplot.hspace**: 0.2  # the amount of height reserved for white space between subplots

### IMAGES

- **image.aspect**: equal  # equal | auto | a number
- **image.interpolation**: bilinear  # see help(imshow) for options
- **image.cmap**: jet  # gray | jet etc...
- **image.cmap**: jet  # the size of the colormap lookup table
- **image.origin**: upper  # lower | upper
Matplotlib, Release 1.4.3

#image.resample : False

### CONTOUR PLOTS
#contour.negative_linestyle : dashed # dashed | solid

### Agg rendering
### Warning: experimental, 2008/10/10
#agg.path.chunksize : 0 # 0 to disable; values in the range
# 10000 to 100000 can improve speed slightly
# and prevent an Agg rendering failure
# when plotting very large data sets,
# especially if they are very gappy.
# It may cause minor artifacts, though.
# A value of 20000 is probably a good
# starting point.

### SAVING FIGURES
#path.simplify : True # When True, simplify paths by removing "invisible"
# points to reduce file size and increase rendering
# speed
#path.simplify_threshold : 0.1 # The threshold of similarity below which
# vertices will be removed in the simplification
# process
#path.snap : True # When True, rectilinear axis-aligned paths will be snapped to
# the nearest pixel when certain criteria are met. When False,
# paths will never be snapped.
#path.sketch : None # May be none, or a 3-tuple of the form (scale, length,
# randomness).
# "scale" is the amplitude of the wiggle
# perpendicular to the line (in pixels). "length"
# is the length of the wiggle along the line (in
# pixels). "randomness" is the factor by which
# the length is randomly scaled.
# the default savefig params can be different from the display params
# e.g., you may want a higher resolution, or to make the figure
# background white
#savefig.dpi : 100 # figure dots per inch
#savefig.facecolor : white # figure facecolor when saving
#savefig.edgecolor : white # figure edgecolor when saving
#savefig.format : png # png, ps, pdf, svg
#savefig.bbox : standard # 'tight' or 'standard'.
# 'tight' is incompatible with pipe-based animation
# backends but will workd with temporary file based ones:
# e.g. setting animation.writer to ffmpeg will not work,
# use ffmpeg_file instead
#savefig.pad_inches : 0.1 # Padding to be used when bbox is set to 'tight'
#savefig.jpeg_quality: 95 # when a jpeg is saved, the default quality parameter.
#savefig.directory : ~ # default directory in savefig dialog box,
# leave empty to always use current working directory
#savefig.transparent : False # setting that controls whether figures are saved with a
# transparent background by default

# tk backend params

2.2. Customizing matplotlib
# tk.window_focus : False  # Maintain shell focus for TkAgg

# ps backend params
#ps.papersize : letter  # auto, letter, legal, ledger, A0-A10, B0-B10
#ps.useafm : False  # use of afm fonts, results in small files
#ps.usedistiller : False  # can be: None, ghostscript or xpdf
    # Experimental: may produce smaller files.
    # xpdf intended for production of publication quality files,
    # but requires ghostscript, xpdf and ps2eps
#ps.distiller.res : 6000  # dpi
#ps.fonttype : 3  # Output Type 3 (Type3) or Type 42 (TrueType)

# pdf backend params
#pdf.compression : 6  # integer from 0 to 9
    # 0 disables compression (good for debugging)
#pdf.fonttype : 3  # Output Type 3 (Type3) or Type 42 (TrueType)

# svg backend params
#svg.image_inline : True  # write raster image data directly into the svg file
#svg.image_noscale : False  # suppress scaling of raster data embedded in SVG
#svg.fonttype : 'path'  # How to handle SVG fonts:
    # 'none': Assume fonts are installed on the machine where the SVG will be viewed.
    # 'path': Embed characters as paths -- supported by most SVG renderers
    # 'svgfont': Embed characters as SVG fonts -- supported only by Chrome,
    # Opera and Safari

# docstring params
#docstring.hardcopy = False  # set this when you want to generate hardcopy docstring

# Set the verbose flags. This controls how much information
# matplotlib gives you at runtime and where it goes. The verbosity
# levels are: silent, helpful, debug, debug-annoying. Any level is
# inclusive of all the levels below it. If your setting is "debug",
# you'll get all the debug and helpful messages. When submitting
# problems to the mailing-list, please set verbose to "helpful" or "debug"
# and paste the output into your report.
#
# The "fileo" gives the destination for any calls to verbose.report.
# These objects can a filename, or a filehandle like sys.stdout.
#
# You can override the rc default verbosity from the command line by
# giving the flags --verbose-LEVEL where LEVEL is one of the legal
# levels, e.g., --verbose-helpful.
#
# You can access the verbose instance in your code
# from matplotlib import verbose.
#verbose.level : silent  # one of silent, helpful, debug, debug-annoying
#verbose.fileo : sys.stdout  # a log filename, sys.stdout or sys.stderr

# Event keys to interact with figures/plots via keyboard.
# Customize these settings according to your needs.
# Leave the field(s) empty if you don't need a key-map. (i.e., fullscren : '')
#keymap.fullscreen : f  # toggling
#keymap.home : h, r, home  # home or reset mnemonic
#keymap.back : left, c, backspace  # forward / backward keys to enable
#keymap.forward : right, v  # left handed quick navigation
#keymap.pan : p  # pan mnemonic
#keymap.zoom : o  # zoom mnemonic
#keymap.save : s  # saving current figure
#keymap.quit : ctrl+w, cmd+w  # close the current figure
#keymap.grid : g  # switching on/off a grid in current axes
#keymap.yscale : l  # toggle scaling of y-axes ('log'/'linear')
#keymap.xscale : L, k  # toggle scaling of x-axes ('log'/'linear')
#keymap.all_axes : a  # enable all axes

# Control location of examples data files
#examples.directory : ''  # directory to look in for custom installation

###ANIMATION settings
#animation.writer : ffmpeg  # MovieWriter 'backend' to use
#animation.codec : mpeg4  # Codec to use for writing movie
#animation.bitrate: -1  # Controls size/quality tradeoff for movie.
# -1 implies let utility auto-determine
#animation.frame_format: 'png'  # Controls frame format used by temp files
#animation.ffmpeg_path: 'ffmpeg'  # Path to ffmpeg binary. Without full path
# - $PATH is searched
#animation.ffmpeg_args: ''  # Additional arguments to pass to ffmpeg
#animation.avconv_path: 'avconv'  # Path to avconv binary. Without full path
# - $PATH is searched
#animation.avconv_args: ''  # Additional arguments to pass to avconv
#animation.mencoder_path: 'mencoder'  # Path to mencoder binary. Without full path
# - $PATH is searched
#animation.mencoder_args: ''  # Additional arguments to pass to mencoder
#animation.convert_path: 'convert'  # Path to ImageMagick's convert binary.
# - On Windows use the full path since convert is also the name of a system tool.

## 2.3 Using matplotlib in a python shell

By default, matplotlib defers drawing until the end of the script because drawing can be an expensive operation, and you may not want to update the plot every time a single property is changed, only once after all the properties have changed.

But when working from the python shell, you usually do want to update the plot with every command, e.g., after changing the xlabel(), or the marker style of a line. While this is simple in concept, in practice it can be tricky, because matplotlib is a graphical user interface application under the hood, and there are some tricks to make the applications work right in a python shell.

### 2.3.1 IPython to the rescue
**Note:** The mode described here still exists for historical reasons, but it is highly advised not to use. It pollutes namespaces with functions that will shadow python built-in and can lead to hard to track bugs. To get IPython integration without imports the use of the `%matplotlib` magic is preferred. See ipython documentation.

Fortunately, ipython, an enhanced interactive python shell, has figured out all of these tricks, and is matplotlib aware, so when you start ipython in the `pylab` mode.

johnh@flag:~> ipython
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.

In [1]: %pylab

    Welcome to pylab, a matplotlib-based Python environment.
    For more information, type `help(pylab)`.

In [2]: x = randn(10000)

In [3]: hist(x, 100)

it sets everything up for you so interactive plotting works as you would expect it to. Call `figure()` and a figure window pops up, call `plot()` and your data appears in the figure window.

Note in the example above that we did not import any matplotlib names because in pylab mode, ipython will import them automatically. ipython also turns on interactive mode for you, which causes every pyplot command to trigger a figure update, and also provides a matplotlib aware run command to run matplotlib scripts efficiently. ipython will turn off interactive mode during a run command, and then restore the interactive state at the end of the run so you can continue tweaking the figure manually.

There has been a lot of recent work to embed ipython, with pylab support, into various GUI applications, so check on the ipython mailing list for the latest status.

### 2.3.2 Other python interpreters

If you can’t use ipython, and still want to use matplotlib/pylab from an interactive python shell, e.g., the plain-ole standard python interactive interpreter, you are going to need to understand what a matplotlib backend is *What is a backend?*.

With the TkAgg backend, which uses the Tkinter user interface toolkit, you can use matplotlib from an arbitrary non-gui python shell. Just set your backend : TkAgg and interactive : True in your matplotlibrc file (see *Customizing matplotlib*) and fire up python. Then:

```python
>>> from pylab import *
>>> plot([1,2,3])
>>> xlabel('hi mom')
```

should work out of the box. This is also likely to work with recent versions of the qt4agg and gtkagg backends, and with the macosx backend on the Macintosh. Note, in batch mode, i.e. when making figures from scripts, interactive mode can be slow since it redraws the figure with each command. So you may want
to think carefully before making this the default behavior via the matplotlibrc file instead of using the functions listed in the next section.

Gui shells are at best problematic, because they have to run a mainloop, but interactive plotting also involves a mainloop. Ipython has sorted all this out for the primary matplotlib backends. There may be other shells and IDEs that also work with matplotlib in interactive mode, but one obvious candidate does not: the python IDLE IDE is a Tkinter gui app that does not support pylab interactive mode, regardless of backend.

### 2.3.3 Controlling interactive updating

The *interactive* property of the pyplot interface controls whether a figure canvas is drawn on every pyplot command. If *interactive* is *False*, then the figure state is updated on every plot command, but will only be drawn on explicit calls to `draw()`. When *interactive* is *True*, then every pyplot command triggers a draw.

The pyplot interface provides 4 commands that are useful for interactive control.

* `isinteractive()` returns the interactive setting *True*|*False*
* `ion()` turns interactive mode on
* `ioff()` turns interactive mode off
* `draw()` forces a figure redraw

When working with a big figure in which drawing is expensive, you may want to turn matplotlib’s interactive setting off temporarily to avoid the performance hit:

```python
>>> #create big-expensive-figure
>>> ioff()  # turn updates off
>>> title('now how much would you pay?')
>>> xticklabels(fontsize=20, color='green')
>>> draw()  # force a draw
>>> savefig('alldone', dpi=300)
>>> close()
>>> ion()   # turn updating back on
>>> plot(rand(20), mfc='g', mec='r', ms=40, mew=4, ls='--', lw=3)
```
3.1 Pyplot tutorial

matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., create a figure, create a plotting area in a figure, plot some lines in a plotting area, decorate the plot with labels, etc.... matplotlib.pyplot is stateful, in that it keeps track of the current figure and plotting area, and the plotting functions are directed to the current axes.

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```
You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the `plot()` command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3].

`plot()` is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```python
plt.plot([1,2,3,4], [1,4,9,16])
```

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is ‘b-’, which is a solid blue line. For example, to plot the above with red circles, you would issue

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```
See the `plot()` documentation for a complete list of line styles and format strings. The `axis()` command in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use `numpy` arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates a plotting several lines with different format styles in one command using arrays.

```python
import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
import numpy as np
import matplotlib.pyplot as plt

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```
### 3.1.1 Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see `matplotlib.lines.Line2D`. There are several ways to set line properties

- Use keyword args:
  ```python
  plt.plot(x, y, linewidth=2.0)
  ```

- Use the setter methods of the Line2D instance. `plot` returns a list of lines; e.g., `line1, line2 = plot(x1, y1, x2, y2)`. Below I have only one line so it is a list of length 1. I use tuple unpacking in the line, = `plot(x, y, 'o')` to get the first element of the list:
  ```python
  line, = plt.plot(x, y, '-
  line.set_antialiased(False) # turn off antialiasing
  ```

- Use the `setp()` command. The example below uses a MATLAB-style command to set multiple properties on a list of lines. `setp` works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:
  ```python
  lines = plt.plot(x1, y1, x2, y2)
  # use keyword args
  plt.setp(lines, color='r', linewidth=2.0)
  ```
Here are the available Line2D properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transform.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>a Path instance and a Transform instance, a Patch</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>the hit testing function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>(np.array xdata, np.array ydata)</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>label</td>
<td>any string</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>['+'</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[ None</td>
</tr>
<tr>
<td>picker</td>
<td>used in interactive line selection</td>
</tr>
<tr>
<td>pickradius</td>
<td>the line pick selection radius</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>np.array</td>
</tr>
<tr>
<td>ydata</td>
<td>np.array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

To get a list of settable line properties, call the `setp()` function with a line or lines as argument

```
In [69]: lines = plt.plot([1,2,3])

In [70]: plt.setp(lines)
    alpha: float
    animated: [True | False]
    antialiased or aa: [True | False]
    ...snip
```
3.1.2 Working with multiple figures and axes

MATLAB, and *pyplot*, have the concept of the current figure and the current axes. All plotting commands apply to the current axes. The function `gca()` returns the current axes (a *matplotlib.axes.Axes* instance), and `gcf()` returns the current figure (*matplotlib.figure.Figure* instance). Normally, you don’t have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

```python
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

The `figure()` command here is optional because `figure(1)` will be created by default, just as a
subplot(111) will be created by default if you don’t manually specify an axes. The subplot() command specifies numrows, numcols, fignum where fignum ranges from 1 to numrows*numcols. The commas in the subplot command are optional if numrows*numcols<10. So subplot(211) is identical to subplot(2,1,1). You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e., not on a rectangular grid, use the axes() command, which allows you to specify the location as axes([left, bottom, width, height]) where all values are in fractional (0 to 1) coordinates. See pylab_examples example code: axes_demo.py for an example of placing axes manually and pylab_examples example code: subplots_demo.py for an example with lots-o-subplots.

You can create multiple figures by using multiple figure() calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```python
import matplotlib.pyplot as plt
plt.figure(1)  # the first figure
plt.subplot(211)  # the first subplot in the first figure
plt.plot([1,2,3])
plt.subplot(212)  # the second subplot in the first figure
plt.plot([4,5,6])
plt.figure(2)  # a second figure
plt.plot([4,5,6])  # creates a subplot(111) by default
plt.figure(1)  # figure 1 current; subplot(212) still current
plt.subplot(211)  # make subplot(211) in figure1 current
plt.title('Easy as 1,2,3')  # subplot 211 title
```

You can clear the current figure with clf() and the current axes with cla(). If you find this statefulness, annoying, don’t despair, this is just a thin stateful wrapper around an object oriented API, which you can use instead (see Artist tutorial)

If you are making a long sequence of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with close(). Deleting all references to the figure, and/or using the window manager to kill the window in which the figure appears on the screen, is not enough, because pyplot maintains internal references until close() is called.

### 3.1.3 Working with text

The text() command can be used to add text in an arbitrary location, and the xlabel(), ylabel() and title() are used to add text in the indicated locations (see Text introduction for a more detailed example)

```python
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
```

3.1. Pyplot tutorial
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100, \ \sigma=15$')
plt.axis([40, 160, 0, .03])
plt.grid(True)
plt.show()

All of the `text()` commands return an `matplotlib.text.Text` instance. Just as with with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp()`:

t = plt.xlabel('my data', fontsize=14, color='red')

These properties are covered in more detail in Text properties and layout.

Using mathematical expressions in text

`matplotlib` accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title, you can write a TeX expression surrounded by dollar signs:

plt.title(r'$\sigma_i=15$')

The `r` preceding the title string is important – it signifies that the string is a raw string and not to treat backslashes as python escapes. `matplotlib` has a built-in TeX expression parser and layout engine, and ships its own math fonts – for details see Writing mathematical expressions. Thus you can use mathematical text
across platforms without requiring a TeX installation. For those who have LaTeX and dvipng installed, you can also use LaTeX to format your text and incorporate the output directly into your display figures or saved postscript – see *Text rendering With LaTeX*.

### Annotating text

The uses of the basic `text()` command above place text at an arbitrary position on the Axes. A common use case of text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x,y)` tuples.

```python
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
             )

plt.ylim(-2,2)
plt.show()
```
In this basic example, both the $xy$ (arrow tip) and $xytext$ locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose – see Annotating text and Annotating Axes for details. More examples can be found in `pylab_examples example code: annotation_demo.py`.

### 3.2 Customizing plots with style sheets

The `style` package adds support for easy-to-switch plotting “styles” with the same parameters as a `matplotlibrc` file.

There are a number of pre-defined styles provided by matplotlib. For example, there’s a pre-defined style called “ggplot”, which emulates the aesthetics of `ggplot` (a popular plotting package for `R`). To use this style, just add:

```python
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
```

To list all available styles, use:

```python
>>> print plt.style.available
```
3.2.1 Defining your own style

You can create custom styles and use them by calling `style.use` with the path or URL to the style sheet. Alternatively, if you add your `<style-name>.mplstyle` file to `mpl_configdir/stylelib`, you can reuse your custom style sheet with a call to `style.use(<style-name>).` By default `mpl_configdir` should be `~/.config/matplotlib`, but you can check where yours is with `matplotlib.get_configdir()`, you may need to create this directory. Note that a custom style sheet in `mpl_configdir/stylelib` will override a style sheet defined by matplotlib if the styles have the same name.

For example, you might want to create `mpl_configdir/stylelib/presentation.mplstyle` with the following:

```
axes.titlesize : 24
axes.labelsize : 20
lines.linewidth : 3
lines.markersize : 10
xtick.labelsize : 16
ytick.labelsize : 16
```

Then, when you want to adapt a plot designed for a paper to one that looks good in a presentation, you can just add:

```python
>>> import matplotlib.pyplot as plt
>>> plt.style.use('presentation')
```

3.2.2 Composing styles

Style sheets are designed to be composed together. So you can have a style sheet that customizes colors and a separate style sheet that alters element sizes for presentations. These styles can easily be combined by passing a list of styles:

```python
>>> import matplotlib.pyplot as plt
>>> plt.style.use(['dark_background', 'presentation'])
```

Note that styles further to the right will overwrite values that are already defined by styles on the left.

3.2.3 Temporary styling

If you only want to use a style for a specific block of code but don’t want to change the global styling, the style package provides a context manager for limiting your changes to a specific scope. To isolate the your styling changes, you can write something like the following:

```python
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> with plt.style.context(('dark_background')):
    plt.plot(np.sin(np.linspace(0, 2*np.pi)), 'r-o')

>>> # Some plotting code with the default style
```
3.3 Interactive navigation

All figure windows come with a navigation toolbar, which can be used to navigate through the data set. Here is a description of each of the buttons at the bottom of the toolbar.

The **Forward and Back buttons**  These are akin to the web browser forward and back buttons. They are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click Back on your web browser before visiting a new page —nothing happens. Home always takes you to the first, default view of your data. For Home, Forward and Back, think web browser where data views are web pages. Use the pan and zoom to rectangle to define new views.

The **Pan/Zoom button**  This button has two modes: pan and zoom. Click the toolbar button to activate panning and zooming, then put your mouse somewhere over an axes. Press the left mouse button and hold it to pan the figure, dragging it to a new position. When you release it, the data under the point where you pressed will be moved to the point where you released. If you press ‘x’ or ‘y’ while panning the motion will be constrained to the x or y axis, respectively. Press the right mouse button to zoom, dragging it to a new position. The x axis will be zoomed in proportionate to the rightward movement and zoomed out proportionate to the leftward movement. Ditto for the y axis and up/down motions. The point under your mouse when you begin the zoom remains stationary, allowing you to zoom to an arbitrary point in the figure. You can use the modifier keys ‘x’, ‘y’ or ‘CONTROL’ to constrain the zoom to the x axis, the y axis, or aspect ratio preserve, respectively.

With polar plots, the pan and zoom functionality behaves differently. The radius axis labels can be dragged using the left mouse button. The radius scale can be zoomed in and out using the right mouse button.

The **Zoom-to-rectangle button**  Click this toolbar button to activate this mode. Put your mouse somewhere over and axes and press the left mouse button. Drag the mouse while holding the button to a new location and release. The axes view limits will be zoomed to the rectangle you have defined. There is also an experimental ‘zoom out to rectangle’ in this mode with the right button, which will place your entire axes in the region defined by the zoom out rectangle.
The **Subplot-configuration button**  Use this tool to configure the parameters of the subplot: the left, right, top, bottom, space between the rows and space between the columns.

The **Save button**  Click this button to launch a file save dialog. You can save files with the following extensions: png, ps, eps, svg and pdf.

### 3.3.1 Navigation Keyboard Shortcuts

The following table holds all the default keys, which can be overwritten by use of your matplotlibrc (#keymap.*).

<table>
<thead>
<tr>
<th>Command</th>
<th>Keyboard Shortcut(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home/Reset</td>
<td>h or r or home</td>
</tr>
<tr>
<td>Back</td>
<td>c or left arrow or backspace</td>
</tr>
<tr>
<td>Forward</td>
<td>v or right arrow</td>
</tr>
<tr>
<td>Pan/Zoom</td>
<td>p</td>
</tr>
<tr>
<td>Zoom-to-rect</td>
<td>o</td>
</tr>
<tr>
<td>Save</td>
<td>ctrl + s</td>
</tr>
<tr>
<td>Toggle fullscreen</td>
<td>ctrl + f</td>
</tr>
<tr>
<td>Close plot</td>
<td>ctrl + w</td>
</tr>
<tr>
<td>Constrain pan/zoom to x axis</td>
<td>hold x when panning/zooming with mouse</td>
</tr>
<tr>
<td>Constrain pan/zoom to y axis</td>
<td>hold y when panning/zooming with mouse</td>
</tr>
<tr>
<td>Preserve aspect ratio</td>
<td>hold CONTROL when panning/zooming with mouse</td>
</tr>
<tr>
<td>Toggle grid</td>
<td>g when mouse is over an axes</td>
</tr>
<tr>
<td>Preserve aspect ratio</td>
<td>L or k when mouse is over an axes</td>
</tr>
<tr>
<td>Toggle x axis scale (log/linear)</td>
<td>I when mouse is over an axes</td>
</tr>
<tr>
<td>Toggle y axis scale (log/linear)</td>
<td></td>
</tr>
</tbody>
</table>

If you are using matplotlib.pyplot the toolbar will be created automatically for every figure. If you are writing your own user interface code, you can add the toolbar as a widget. The exact syntax depends on your UI, but we have examples for every supported UI in the matplotlib/examples/user_interfaces directory. Here is some example code for GTK:

```python
from matplotlib.figure import Figure
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as NavigationToolbar

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
```
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

text = gtk.VBox()
win.add(text)

fig = Figure(figsize=(5, 4), dpi=100)
ax = fig.add_subplot(111)
ax.plot([1, 2, 3])

canvas = FigureCanvas(fig)  # a gtk.DrawingArea
toolbar = NavigationToolbar(canvas, win)

vbox.pack_start(canvas)
vbox.pack_start(toolbar, False, False)

win.show_all()
gtk.main()

3.4 Working with text

3.4.1 Text introduction

matplotlib has excellent text support, including mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. Because we embed the fonts directly in the output documents, e.g., for postscript or PDF, what you see on the screen is what you get in the hardcopy. freetype2 support produces very nice, antialiased fonts, that look good even at small raster sizes. matplotlib includes its own matplotlib.font_manager, thanks to Paul Barrett, which implements a cross platform, W3C compliant font finding algorithm.

You have total control over every text property (font size, font weight, text location and color, etc) with sensible defaults set in the rc file. And significantly for those interested in mathematical or scientific figures, matplotlib implements a large number of TeX math symbols and commands, to support mathematical expressions anywhere in your figure.

3.4.2 Basic text commands

The following commands are used to create text in the pyplot interface

- **text()** - add text at an arbitrary location to the Axes; matplotlib.axes.Axes.text() in the API.
- **xlabel()** - add an axis label to the x-axis; matplotlib.axes.Axes.set_xlabel() in the API.
- **ylabel()** - add an axis label to the y-axis; matplotlib.axes.Axes.set_ylabel() in the API.
- **title()** - add a title to the Axes; matplotlib.axes.Axes.set_title() in the API.
- **figtext()** - add text at an arbitrary location to the Figure; matplotlib.figure.Figure.text() in the API.
- **suptitle()** - add a title to the Figure; matplotlib.figure.Figure.suptitle() in the API.
- **annotate()** - add an annotation, with optional arrow, to the Axes; matplotlib.axes.Axes.annotate() in the API.

All of these functions create and return a matplotlib.text.Text() instance, which can be configured with a variety of font and other properties. The example below shows all of these commands in action.

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')
ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')
ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
    bbox={'facecolor':'red', 'alpha':0.5, 'pad':10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, unicode('unicode: Institut f\374r Festk\366rperphysik', 'latin-1'))

ax.text(0.95, 0.01, 'colored text in axes coords',
    verticalalignment='bottom', horizontalalignment='right',
    transform=ax.transAxes,
    color='green', fontsize=15)

ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
    arrowprops=dict(facecolor='black', shrink=0.05))

ax.axis([0, 10, 0, 10])

plt.show()
```
3.4.3 Text properties and layout

The `matplotlib.text.Text` instances have a variety of properties which can be configured via keyword arguments to the text commands (e.g., `title()`, `xlabel()` and `text()`).
### Property Value Type

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict plus key 'pad' which is a pad in points</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transform.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>a Path instance and a Transform instance, a Patch</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>family</td>
<td>['serif'</td>
</tr>
<tr>
<td>fontproperties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>['center'</td>
</tr>
<tr>
<td>label</td>
<td>any string</td>
</tr>
<tr>
<td>linespacing</td>
<td>float</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left'</td>
</tr>
<tr>
<td>name or fontname</td>
<td>string e.g., ['Sans'</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees 'vertical'</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[ size in points</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>['normal'</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transform transformation instance</td>
</tr>
<tr>
<td>variant</td>
<td>['normal'</td>
</tr>
<tr>
<td>verticalalignment or va</td>
<td>['center'</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>['normal'</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

You can layout text with the alignment arguments `horizontalalignment`, `verticalalignment`, and `multialignment`. `horizontalalignment` controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. `verticalalignment` controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. `multialignment`, for newline separated strings only, controls whether the different lines are left, center or right justified. Here is an example which uses the `text()` command to show the various alignment possibilities. The use of `transform=ax.transAxes` throughout the code indicates that the coordinates are given relative to the axes bounding box, with 0,0 being the lower left of the axes and 1,1 the upper right.

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height
```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

# axes coordinates are 0,0 is bottom left and 1,1 is upper right
p = patches.Rectangle(
    (left, bottom), width, height,
    fill=False, transform=ax.transAxes, clip_on=False
)

ax.add_patch(p)
ax.text(left, bottom, 'left top',
    horizontalalignment='left',
    verticalalignment='top',
    transform=ax.transAxes)
ax.text(left, bottom, 'left bottom',
    horizontalalignment='left',
    verticalalignment='bottom',
    transform=ax.transAxes)
ax.text(right, top, 'right bottom',
    horizontalalignment='right',
    verticalalignment='bottom',
    transform=ax.transAxes)
ax.text(right, top, 'right top',
    horizontalalignment='right',
    verticalalignment='top',
    transform=ax.transAxes)
ax.text(right, bottom, 'center top',
    horizontalalignment='center',
    verticalalignment='top',
    transform=ax.transAxes)
ax.text(left, 0.5*(bottom+top), 'right center',
    horizontalalignment='right',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)
ax.text(left, 0.5*(bottom+top), 'left center',
    horizontalalignment='left',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)
ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
    horizontalalignment='center',
    verticalalignment='center',
    fontsize=20, color='red',
    transform=ax.transAxes)
3.4.4 Writing mathematical expressions

You can use a subset TeX markup in any matplotlib text string by placing it inside a pair of dollar signs ($). Note that you do not need to have TeX installed, since matplotlib ships its own TeX expression parser, layout engine and fonts. The layout engine is a fairly direct adaptation of the layout algorithms in Donald Knuth’s TeX, so the quality is quite good (matplotlib also provides a `usetex` option for those who do want to call out to TeX to generate their text (see *Text rendering With LaTeX*).
Any text element can use math text. You should use raw strings (precede the quotes with an ‘r’), and surround the math text with dollar signs ($), as in TeX. Regular text and mathtext can be interleaved within the same string. Mathtext can use the Computer Modern fonts (from (La)TeX), STIX fonts (with are designed to blend well with Times) or a Unicode font that you provide. The mathtext font can be selected with the customization variable `mathtext.fontset` (see *Customizing matplotlib*).

**Note:** On “narrow” builds of Python, if you use the STIX fonts you should also set `ps.fonttype` and `pdf.fonttype` to 3 (the default), not 42. Otherwise some characters will not be visible.

Here is a simple example:

```python
# plain text
plt.title('alpha > beta')
```

produces “alpha > beta”.

Whereas this:

```python
# math text
plt.title(r'$\alpha > \beta$')
```

produces “α > β”.

**Note:** Mathtext should be placed between a pair of dollar signs ($). To make it easy to display monetary values, e.g., “$100.00”, if a single dollar sign is present in the entire string, it will be displayed verbatim as a dollar sign. This is a small change from regular TeX, where the dollar sign in non-math text would have to be escaped (‘$’).

**Note:** While the syntax inside the pair of dollar signs ($) aims to be TeX-like, the text outside does not. In particular, characters such as:

```python
# $ % & ~ \ ^ _ \ { } \( \) \[ \]
```

have special meaning outside of math mode in TeX. Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag. See the *usetex tutorial* for more information.

### Subscripts and superscripts

To make subscripts and superscripts, use the ‘_’ and ‘^’ symbols:

```python
r'$\alpha_i > \beta_i$'
```

\[ \alpha_i > \beta_i \]  \hspace{1cm} (3.1)

Some symbols automatically put their sub/superscripts under and over the operator. For example, to write the sum of \( x_i \) from 0 to \( \infty \), you could do:
Fractions, binomials and stacked numbers

Fractions, binomials and stacked numbers can be created with the \texttt{\frac{\{}{\}}}, \texttt{\binom{\{}{\}} and \texttt{\stackrel{\{}{\}} commands, respectively:

\begin{equation}
\frac{3}{4} \binom{3}{4} \stackrel{3}{4}
\end{equation}

produces

\begin{equation}
\frac{3}{4} \binom{3}{4} \stackrel{3}{4}
\end{equation}

Fractions can be arbitrarily nested:

\begin{equation}
\frac{5 - \frac{1}{x}}{4}
\end{equation}

produces

\begin{equation}
\frac{5 - \frac{1}{x}}{4}
\end{equation}

Note that special care needs to be taken to place parentheses and brackets around fractions. Doing things the obvious way produces brackets that are too small:

\begin{equation}
\left(\frac{5 - \frac{1}{x}}{4}\right)
\end{equation}

Radicals

Radicals can be produced with the \texttt{\sqrt[\{}{\}} command. For example:

\begin{equation}
\sqrt{2}
\end{equation}
\sqrt{2} \quad (3.7)

Any base can (optionally) be provided inside square brackets. Note that the base must be a simple expression, and can not contain layout commands such as fractions or sub/superscripts:

```latex
r'\sqrt[3]{x}'
```

\sqrt[3]{x} \quad (3.8)

**Fonts**

The default font is *italics* for mathematical symbols.

---

**Note:** This default can be changed using the `mathtext.default` rcParam. This is useful, for example, to use the same font as regular non-math text for math text, by setting it to *regular*.

---

To change fonts, e.g., to write “sin” in a Roman font, enclose the text in a font command:

```latex
r'\mathcal{A}\sin(2 \omega t)'
```

\[ s(t) = A\sin(2\omega t) \quad (3.9) \]

More conveniently, many commonly used function names that are typeset in a Roman font have shortcuts. So the expression above could be written as follows:

```latex
r'\mathcal{A}\sin(2 \omega t)'
```

\[ s(t) = A\sin(2\omega t) \quad (3.10) \]

Here “s” and “t” are variable in italics font (default), “sin” is in Roman font, and the amplitude “A” is in calligraphy font. Note in the example above the calligraphy \(\mathcal{A}\) is squished into the \(\sin\). You can use a spacing command to add a little whitespace between them:

```latex
s(t) = A\sin(2\omega t)
```

\[ s(t) = \mathcal{A}\sin(2\omega t) \quad (3.11) \]

The choices available with all fonts are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\text{Roman}</code></td>
<td>Roman</td>
</tr>
<tr>
<td><code>\text{Italic}</code></td>
<td><em>Italic</em></td>
</tr>
<tr>
<td><code>\text{Typewriter}</code></td>
<td>Typewriter</td>
</tr>
<tr>
<td><code>\text{CALLIGRAPHY}</code></td>
<td><em>CALLIGRAPHY</em></td>
</tr>
</tbody>
</table>

When using the STIX fonts, you also have the choice of:
There are also three global “font sets” to choose from, which are selected using the `mathtext.fontset` parameter in `matplotlibrc`.

cm: Computer Modern (TeX)

\[
\mathcal{R} \prod_{i=\alpha_i+1}^{\infty} a_i \sin(2\pi f x_i)
\]

stix: STIX (designed to blend well with Times)

\[
\mathcal{R} \prod_{i=\alpha_i+1}^{\infty} a_i \sin(2\pi f x_i)
\]

stixsans: STIX sans-serif

\[
\mathcal{R} \prod_{i=\alpha_i+1}^{\infty} a_i \sin(2\pi f x_i)
\]

Additionally, you can use \texttt{\textbackslash mathdefault{...}} or its alias \texttt{\textbackslash mathregular{...}} to use the font used for regular text outside of mathtext. There are a number of limitations to this approach, most notably that far fewer symbols will be available, but it can be useful to make math expressions blend well with other text in the plot.

### Custom fonts

Matplotlib also provides a way to use custom fonts for math. This method is fairly tricky to use, and should be considered an experimental feature for patient users only. By setting the rcParam `mathtext.fontset` to `custom`, you can then set the following parameters, which control which font file to use for a particular set of math characters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Corresponds to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mathtext.it</code></td>
<td>\textbackslash mathit{} or default italic</td>
</tr>
<tr>
<td><code>mathtext.rm</code></td>
<td>\textbackslash mathrm{} Roman (upright)</td>
</tr>
<tr>
<td><code>mathtext.tt</code></td>
<td>\textbackslash math\texttt{} Typewriter (monospace)</td>
</tr>
<tr>
<td><code>mathtext.bf</code></td>
<td>\textbackslash math\textbf{} bold italic</td>
</tr>
<tr>
<td><code>mathtext.cal</code></td>
<td>\textbackslash math\textcal{} calligraphic</td>
</tr>
<tr>
<td><code>mathtext.sf</code></td>
<td>\textbackslash math\textsf{} sans-serif</td>
</tr>
</tbody>
</table>
Each parameter should be set to a fontconfig font descriptor (as defined in the yet-to-be-written font chapter).

The fonts used should have a Unicode mapping in order to find any non-Latin characters, such as Greek. If you want to use a math symbol that is not contained in your custom fonts, you can set the rcParam `mathtext.fallback_to_cm` to `True` which will cause the mathtext system to use characters from the default Computer Modern fonts whenever a particular character can not be found in the custom font.

Note that the math glyphs specified in Unicode have evolved over time, and many fonts may not have glyphs in the correct place for mathtext.

### Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\acute{a} or 'a</td>
<td>ȧ</td>
</tr>
<tr>
<td>\bar{a}</td>
<td>ţ</td>
</tr>
<tr>
<td>\breve{a}</td>
<td>ă</td>
</tr>
<tr>
<td>\ddot{a} or &quot;a</td>
<td>ȧ̂</td>
</tr>
<tr>
<td>\dot{a} or .a</td>
<td>ȧ̈</td>
</tr>
<tr>
<td>\grave{a} or \‘a</td>
<td>ǡ</td>
</tr>
<tr>
<td>\hat{a} or ^a</td>
<td>ȧ̂</td>
</tr>
<tr>
<td>\tilde{a} or ~a</td>
<td>ȧ̃</td>
</tr>
<tr>
<td>\vec{a}</td>
<td>ȧ̃</td>
</tr>
<tr>
<td>\overline{abc}</td>
<td>abc̄</td>
</tr>
</tbody>
</table>

In addition, there are two special accents that automatically adjust to the width of the symbols below:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\widehat{xyz}</td>
<td>̂xyz</td>
</tr>
<tr>
<td>\widetilde{xyz}</td>
<td>̃xyz</td>
</tr>
</tbody>
</table>

Care should be taken when putting accents on lower-case i’s and j’s. Note that in the following `\imath` is used to avoid the extra dot over the i:

```latex
$r"\hat i \ \ \hat \imath$"
```

\[\hat i \ \ \hat \imath \]  \hspace{1cm} (3.12)

### Symbols

You can also use a large number of the TeX symbols, as in `\infty`, `\leftarrow`, `\sum`, `\int`.

#### Lower-case Greek
Upper-case Greek

<table>
<thead>
<tr>
<th>$\Delta \Delta$</th>
<th>$\Gamma \Gamma$</th>
<th>$\Lambda \Lambda$</th>
<th>$\Omega \Omega$</th>
<th>$\Phi \Phi$</th>
<th>$\Pi \Pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Psi \Psi$</td>
<td>$\Sigma \Sigma$</td>
<td>$\Theta \Theta$</td>
<td>$\Upsilon \Upsilon$</td>
<td>$\Xi \Xi$</td>
<td>$\Upsilon \Upsilon$</td>
</tr>
<tr>
<td>$\nabla \nabla$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hebrew

| $\aleph \aleph$ | $\beth \beth$ | $\daleth \daleth$ | $\gimel \gimel$ |

Delimiters

<table>
<thead>
<tr>
<th>//</th>
<th>[[</th>
<th>↓ \Downarrow</th>
<th>↑ \Uparrow</th>
<th>\Vert \Vert</th>
<th>\backslash \backslash</th>
</tr>
</thead>
<tbody>
<tr>
<td>\downarrow</td>
<td>\langle</td>
<td>\lceil</td>
<td>\rceil</td>
<td>\lfloor</td>
<td>\llcorner</td>
</tr>
<tr>
<td>\rangle</td>
<td>{</td>
<td>|</td>
<td>}</td>
<td>\urcorner</td>
<td>\uparrow</td>
</tr>
<tr>
<td>\vert</td>
<td>{{</td>
<td>||</td>
<td>}}</td>
<td></td>
<td>\urcorner</td>
</tr>
</tbody>
</table>

Big symbols

<table>
<thead>
<tr>
<th>\bigcap</th>
<th>\bigcup</th>
<th>\bigodot</th>
<th>\bigoplus</th>
<th>\bigotimes</th>
<th>\ prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>\biguplus</td>
<td>\bigvee</td>
<td>\bigwedge</td>
<td>\coprod</td>
<td>\int</td>
<td></td>
</tr>
</tbody>
</table>

Standard function names

<table>
<thead>
<tr>
<th>$\Pr \Pr$</th>
<th>$\arccos \arccos$</th>
<th>$\arcsin \arcsin$</th>
<th>$\arctan \arctan$</th>
<th>$\cot \cot$</th>
<th>$\det \det$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\arg \arg$</td>
<td>$\cos \cos$</td>
<td>$\cosh \cosh$</td>
<td>$\deg \deg$</td>
<td>$\hom \hom$</td>
<td>$\lim \lim$</td>
</tr>
<tr>
<td>$\coth \coth$</td>
<td>$\csc \csc$</td>
<td>$\gcd \gcd$</td>
<td>$\lg \lg$</td>
<td>$\log \log$</td>
<td>$\sin \sin$</td>
</tr>
<tr>
<td>$\dim \dim$</td>
<td>$\exp \exp$</td>
<td>$\ker \ker$</td>
<td></td>
<td>$\sinh \tanh$</td>
<td></td>
</tr>
<tr>
<td>$\inf \inf$</td>
<td>$\ker \ker$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\liminf \liminf$</td>
<td>$\limsup \limsup$</td>
<td>$\operatorname{min} \min$</td>
<td>$\operatorname{tan} \tan$</td>
<td>$\tanh \tanh$</td>
<td></td>
</tr>
<tr>
<td>$\max \max$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sinh \sinh$</td>
<td>$\sup \sup$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Binary operation and relation symbols
### Arrow Symbols

| \trianglelefteq | \ntrianglelefteq | \trianglerighteq | \ntrianglerighteq | \nvdash | \vdash | ⊙ | \odot | \ominus | \oplus | \oslash | \otimes | ‖ | \parallel | ⊥ | \perp | ⊭ | \vDash | \vdash | \sim | \simeq | / | \slash | ⌢ | \smile | ⊓ | \sqcap | ⊔ | \sqcup | ⊏ | \sqsubset | ⊏ | \sqsubset | ⪯ | \preceq | ⪹ | \precnapprox | ⪨ | \precnsim | ≼ | \preccurlyeq | ⪯ | \succeq | ⪺ | \succnapprox | ⪩ | \succsim | ⊃ | \supset | ⊇ | \supseteq | ⫆ | \supseteqq | ⪉ | \supsetneq | ⫌ | \supsetneqq | ∴ | \therefore | × | \times | ⊤ | \top | \triangleleft | \trianglelefteq | \triangleright | \trianglerighteq | \uplus | \vDash | \varpropto | \vartriangleleft | \vartriangleright | \vdash | \veebar | \wedge | \wr

| \triangleq | \ntriangleq | \triangleright | \trianglerighteq | \uplus | \vartrianglerighteq | \vartriangleright | \vdash | \vee | \wr

| \Downarrow | \Leftarrow | \Rightarrow |
| \Leftarrow | \Leftarrow | \Leftarrow |
| \Longleftarrow | \Longleftarrow | \Longleftarrow |
| \Rightarrow | \Rightarrow | \Rightarrow |
| \Lsh | \Lsh | \Lsh |
| \Nwarrow | \Nwarrow | \Nwarrow |
| \Lleftarrow | \Lleftarrow | \Lleftarrow |
| \leftharpoonup | \leftharpoonup | \leftharpoonup |
| \curvearrowright | \curvearrowright | \curvearrowright |
| \dashleftarrow | \dashleftarrow | \dashleftarrow |
| \downharpoonleft | \downharpoonleft | \downharpoonleft |
| \hookleftarrow | \hookleftarrow | \hookleftarrow |
| \Arrowleft | \Arrowleft | \Arrowleft |
| \Leftarrowtail | \Leftarrowtail | \Leftarrowtail |
| \leftharpoonup | \leftharpoonup | \leftharpoonup |
| \leftharpoons | \leftharpoons | \leftharpoons |
| \leftharpoons | \leftharpoons | \leftharpoons |
| \leftsquigarrow | \leftsquigarrow | \leftsquigarrow |
| \leftrightsquigarrow | \leftrightsquigarrow | \leftrightsquigarrow |

#### 3.4. Working with text
If a particular symbol does not have a name (as is true of many of the more obscure symbols in the STIX fonts), Unicode characters can also be used:

\texttt{ur'\$\u23ce$'}
Example

Here is an example illustrating many of these features in context.

```python
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
plt.plot(t,s)
plt.title(r'$\alpha_i > \beta_i$', fontsize=20)
plt.text(1, -0.6, r'$\sum_{i=0}^\infty x_i$', fontsize=20)
plt.text(0.6, 0.6, r'$\mathcal{A}\sin(2 \omega t)$', fontsize=20)
plt.xlabel('time (s)')
plt.ylabel('volts (mV)')
plt.show()
```

3.4.5 Typesetting With XeLaTeX/LuaLaTeX

Using the pgf backend, matplotlib can export figures as pgf drawing commands that can be processed with pdflatex, xelatex or lualatex. XeLaTeX and LuaLaTeX have full unicode support and can use any font that is installed in the operating system, making use of advanced typographic features of OpenType, AAT and Graphite. PgF pictures created by `plt.savefig('figure.pgf')` can be embedded as
raw commands in LaTeX documents. Figures can also be directly compiled and saved to PDF with
plt.savefig('figure.pdf') by either switching to the backend

matplotlib.use('pgf')

or registering it for handling pdf output

from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)

The second method allows you to keep using regular interactive backends and to save xelatex, lualatex or
pdflatex compiled PDF files from the graphical user interface.

Matplotlib’s pgf support requires a recent LaTeX installation that includes the TikZ/PGF packages (such as
TeXLive), preferably with XeLaTeX or LuaLaTeX installed. If either pdftocairo or ghostscript is present on
your system, figures can optionally be saved to PNG images as well. The executables for all applications
must be located on your PATH.

Rc parameters that control the behavior of the pgf backend:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgf.preamble</td>
<td>Lines to be included in the LaTeX preamble</td>
</tr>
<tr>
<td>pgf.rcfonts</td>
<td>Setup fonts from rc params using the fontspec package</td>
</tr>
<tr>
<td>pgf.texsystem</td>
<td>Either “xelatex” (default), “lualatex” or “pdflatex”</td>
</tr>
</tbody>
</table>

Note: TeX defines a set of special characters, such as:

# $ % & _ ^ \ { }

Generally, these characters must be escaped correctly. For convenience, some characters (_,^,%) are auto-
matically escaped outside of math environments.

Font specification

The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined
in the matplotlib rc parameters. You can also use the LaTeX default Computer Modern fonts by clearing
the lists for font.serif, font.sans-serif or font.monospace. Please note that the glyph coverage of
these fonts is very limited. If you want to keep the Computer Modern font face but require extended unicode
support, consider installing the Computer Modern Unicode fonts CMU Serif, CMU Sans Serif, etc.

When saving to .pgf, the font configuration matplotlib used for the layout of the figure is included in the
header of the text file.

# -*- coding: utf-8 -*-

import matplotlib as mpl
mpl.use("pgf")
pgf_with_rc_fonts = {
    "font.family": "serif",
    "font.serif": [],
    "font.sans-serif": ["DejaVu Sans"], # use a specific sans-serif font
    "font.style": "normal",
    "font.weight": "normal",
    "font.size": 10,
    "figure.titlesize": 10,
    "figure.labelsize": 10,
    "figure.ticks": 0,
    "figure.dpi": 100
}

# -- coding: utf-8 --
mpl.rcParams.update(pgf_with_rc_fonts)

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(4.5,2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.text(2.5, 1., "comic sans", family="Comic Sans MS")
plt.xlabel(u"\mu is not $\mu$")
plt.tight_layout(.5)
```

---

**Custom preamble**

Full customization is possible by adding your own commands to the preamble. Use the `pgf.preamble` parameter if you want to configure the math fonts, using `unicode-math` for example, or for loading additional packages. Also, if you want to do the font configuration yourself instead of using the fonts specified in the rc parameters, make sure to disable `pgf.rcfonts`.

```python
# -*- coding: utf-8 -*-
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)
import six
import matplotlib as mpl
mpl.use("pgf")
pgf_with_custom_preamble = {
    "font.family": "serif",  # use serif/main font for text elements
    "text.usetex": True,  # use inline math for ticks
    "pgf.rcfonts": False,  # don't setup fonts from rc parameters
    "pgf.preamble": [  
        "\usepackage{units}",  # load additional packages
        "\usepackage{metalogo}",
        "\usepackage{unicode-math}",  # unicode math setup
        r"\setmathfont{xits-math.otf}"],
}
```

---

3.4. Working with text
Choosing the TeX system

The TeX system to be used by matplotlib is chosen by the `pgf.texsystem` parameter. Possible values are 'xelatex' (default), 'lualatex' and 'pdflatex'. Please note that when selecting pdflatex the fonts and unicode handling must be configured in the preamble.

```python
# -*- coding: utf-8 -*-
import matplotlib as mpl
mpl.use("pgf")
pgf_with_pdflatex = {
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": [
        r"\usepackage[utf8x]{inputenc}",
        r"\usepackage[T1]{fontenc}",
        r"\usepackage[cmbright]{cmap}"
    ]
}
mpl.rcParams.update(pgf_with_pdflatex)

import matplotlib.pyplot as plt
plt.figure(figsize=(4.5,2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif", family="serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.xlabel(u"\mu is not $\mu$")
plt.tight_layout(.5)
```
```latex
\begin{figure}
\centering
\begin{tikzpicture}
    \begin{axis}[
        width=\textwidth,
        height=\textwidth,
        axis lines=middle,
        xlabel=$\mu$ is not $\mu$,
        ylabel=\texttt{serif} \quad \texttt{monospace} \quad \texttt{sans-serif},
        xmin=0, xmax=4,
        ymin=0, ymax=4,
        xtick={0,0.5,1,1.5,2,2.5,3,3.5,4},
        ytick={0,0.5,1,1.5,2,2.5,3,3.5,4},
    ]
    \end{axis}
\end{tikzpicture}
\end{figure}
```

### Troubleshooting

- Please note that the TeX packages found in some Linux distributions and MiKTeX installations are dramatically outdated. Make sure to update your package catalog and upgrade or install a recent TeX distribution.

- On Windows, the `PATH` environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment Variables* and *Setting environment variables in windows* for details.

- A limitation on Windows causes the backend to keep file handles that have been opened by your application open. As a result, it may not be possible to delete the corresponding files until the application closes (see #1324).

- Sometimes the font rendering in figures that are saved to png images is very bad. This happens when the pdftocairo tool is not available and ghostscript is used for the pdf to png conversion.

- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.

- The `pgf.preamble` rc setting provides lots of flexibility, and lots of ways to cause problems. When experiencing problems, try to minimalize or disable the custom preamble.

- Configuring an `unicode-math` environment can be a bit tricky. The TeXLive distribution for example provides a set of math fonts which are usually not installed system-wide. XeTeX, unlike LuaLatex, cannot find these fonts by their name, which is why you might have to specify `\setmathfont{xits-math.otf}` instead of `\setmathfont{XITS Math}` or alternatively make the fonts available to your OS. See this [tex.stackexchange.com question](https://tex.stackexchange.com) for more details.

- If the font configuration used by matplotlib differs from the font setting in your LaTeX document, the alignment of text elements in imported figures may be off. Check the header of your `.pgf` file if you are unsure about the fonts matplotlib used for the layout.

- If you still need help, please see *Getting help*
3.4.6 Text rendering With LaTeX

Matplotlib has the option to use LaTeX to manage all text layout. This option is available with the following backends:

- Agg
- PS
- PDF

The LaTeX option is activated by setting `text.usetex : True` in your rc settings. Text handling with matplotlib’s LaTeX support is slower than matplotlib’s very capable `mathtext`, but is more flexible, since different LaTeX packages (font packages, math packages, etc.) can be used. The results can be striking, especially when you take care to use the same fonts in your figures as in the main document.

Matplotlib’s LaTeX support requires a working LaTeX installation, dvipng (which may be included with your LaTeX installation), and Ghostscript (GPL Ghostscript 8.60 or later is recommended). The executables for these external dependencies must all be located on your PATH.

There are a couple of options to mention, which can be changed using `rc settings`. Here is an example `matplotlibrc` file:

```python
font.family : serif
font.serif : Times, Palatino, New Century Schoolbook, Bookman, Computer Modern Roman
font.sans-serif : Helvetica, Avant Garde, Computer Modern Sans serif
font.cursive : Zapf Chancery
font.monospace : Courier, Computer Modern Typewriter

text.usetex : true
```

The first valid font in each family is the one that will be loaded. If the fonts are not specified, the Computer Modern fonts are used by default. All of the other fonts are Adobe fonts. Times and Palatino each have their own accompanying math fonts, while the other Adobe serif fonts make use of the Computer Modern math fonts. See the PSNFSS documentation for more details.

To use LaTeX and select Helvetica as the default font, without editing `matplotlibrc` use:

```python
from matplotlib import rc
cr('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
## for Palatino and other serif fonts use:
#rc('font',**{'family':serif,'serif':['Palatino']})
rc('text', usetex=True)
```

Here is the standard example, `tex_demo.py`:

```python
""
Demo of TeX rendering.
""

You can use TeX to render all of your matplotlib text if the rc parameter `text.usetex` is set. This works currently on the agg and ps backends, and requires that you have tex and the other dependencies described at http://matplotlib.sf.net/matplotlib.texmanager.html properly installed on your system. The first time you run a script you will see a lot of output from tex and associated tools. The next
time, the run may be silent, as a lot of the information is cached in
~/.tex.cache

```
import numpy as np
import matplotlib.pyplot as plt

# Example data
T = np.arange(0.0, 1.0 + 0.01, 0.01)
S = np.cos(4 * np.pi * T) + 2

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.plot(T, S)
plt.xlabel(r'$\textbf{time}$ (s)')
plt.ylabel(r'$\textit{voltage}$ (mV)', fontsize=16)
plt.title(r'\TeX\ is Number
\[ \sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n} \]\!',
fontsize=16, color='gray')
# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)

plt.savefig('tex_demo')
plt.show()
```
Note that display math mode ($$ e=mc^2 $$) is not supported, but adding the command \displaystyle, as in tex_demo.py, will produce the same results.

Note: Certain characters require special escaping in TeX, such as:

```latex
# $ % & ^ \ { } \( \) \[ \]
```

Therefore, these characters will behave differently depending on the rcParam text.usetex flag.

usetex with unicode

It is also possible to use unicode strings with the LaTeX text manager, here is an example taken from tex_unicode_demo.py:

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
This demo is tex_demo.py modified to have unicode. See that file for more information.
"""

from __future__ import unicode_literals
import matplotlib as mpl
mpl.rcParams['text.usetex']=True
mpl.rcParams['text.latex.unicode']=True
```

Note:

\[
\sum_{n=1}^{\infty} -\frac{e^{i\pi}}{2^n}!
\]
from numpy import arange, cos, pi
from matplotlib.pyplot import (figure, axes, plot, xlabel, ylabel, title, grid, savefig, show)

figure(1, figsize=(6,4))
ax = axes([0.1, 0.1, 0.8, 0.7])
t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*2*pi*t)+2
plot(t, s)
xlabel(r'$\textbf{time (s)}$')
ylabel(r'$\textit{Velocity (°/sec)}$', fontsize=16)
title(r'\TeX\ is Number $\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!', fontsize=16, color=r'r')
grid(True)
show()

\TeX\ is Number $\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!

**Postscript options**

In order to produce encapsulated postscript files that can be embedded in a new \LaTeX\ document, the default behavior of matplotlib is to distill the output, which removes some postscript operators used by \LaTeX\ that are illegal in an eps file. This step produces results which may be unacceptable to some users, because the text is coarsely rasterized and converted to bitmaps, which are not scalable like standard postscript, and the text is not searchable. One workaround is to to set `ps.distiller.res` to a higher value (perhaps 6000) in your rc settings, which will produce larger files but may look better and scale reasonably. A better workaround, which requires Poppler or Xpdf, can be activated by changing the `ps.usedistiller` rc setting.
Possible hangups

- On Windows, the PATH environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See Environment Variables and Setting environment variables in windows for details.

- Using MiKTeX with Computer Modern fonts, if you get odd *Agg and PNG results, go to MiKTeX/Options and update your format files

- The fonts look terrible on screen. You are probably running Mac OS, and there is some funny business with older versions of dvipng on the mac. Set text.dvipnghack : True in your matplotlibrc file.

- On Ubuntu and Gentoo, the base texlive install does not ship with the type1cm package. You may need to install some of the extra packages to get all the goodies that come bundled with other latex distributions.

- Some progress has been made so matplotlib uses the dvi files directly for text layout. This allows latex to be used for text layout with the pdf and svg backends, as well as the *Agg and PS backends. In the future, a latex installation may be the only external dependency.

Troubleshooting

- Try deleting your .matplotlib/tex.cache directory. If you don’t know where to find .matplotlib, see .matplotlib directory location.

- Make sure LaTeX, dvipng and ghostscript are each working and on your PATH.

- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.

- Most problems reported on the mailing list have been cleared up by upgrading Ghostscript. If possible, please try upgrading to the latest release before reporting problems to the list.

- The text.latex.preamble rc setting is not officially supported. This option provides lots of flexibility, and lots of ways to cause problems. Please disable this option before reporting problems to the mailing list.

- If you still need help, please see Getting help

3.4.7 Annotating text

For a more detailed introduction to annotations, see Annotating Axes.

The uses of the basic text() command above place text at an arbitrary position on the Axes. A common use case of text is to annotate some feature of the plot, and the annotate() method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument xy and the location of the text xytext. Both of these arguments are (x,y) tuples.
In this example, both the $xy$ (arrow tip) and $xytext$ locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose – you can specify the coordinate system of $xy$ and $xytext$ with one of the following strings for $xycoords$ and $textcoords$ (default is ‘data’).
Matplotlib, Release 1.4.3

<table>
<thead>
<tr>
<th>argument</th>
<th>coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>'figure points'</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure pixels'</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure fraction'</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>'axes points'</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>'axes pixels'</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>'axes fraction'</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>'data'</td>
<td>use the axes data coordinate system</td>
</tr>
</tbody>
</table>

For example to place the text coordinates in fractional axes coordinates, one could do:

```python
tax.annotate('local max', xy=(3, 1), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top',
            )
```

For physical coordinate systems (points or pixels) the origin is the (bottom, left) of the figure or axes. If
the value is negative, however, the origin is from the (right, top) of the figure or axes, analogous to negative
indexing of sequences.

Optionally, you can specify arrow properties which draws an arrow from the text to the annotated point by
giving a dictionary of arrow properties in the optional keyword argument `arrowprops`.

<table>
<thead>
<tr>
<th>arrowprops key</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>move the tip and base some percent away from the annotated point and text</td>
</tr>
<tr>
<td>**kwargs</td>
<td>any key for matplotlib.patches.Polygon, e.g., facecolor</td>
</tr>
</tbody>
</table>

In the example below, the `xy` point is in native coordinates (xycoords defaults to ‘data’). For a polar
axes, this is in (theta, radius) space. The text in this example is placed in the fractional figure coordinate
system. `matplotlib.text.Text` keyword args like horizontalalignment, verticalalignment and
fontsize are passed from the `-matplotlib.Axes.annotate` to the `Text` instance

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
r = np.arange(0,1,0.001)
theta = 2*np.pi*r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr), # theta, radius
            xytext=(0.05, 0.05), # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            )
```

horizontalalignment='left',
verticalalignment='bottom',
)
plt.show()

For more on all the wild and wonderful things you can do with annotations, including fancy arrows, see
Annotating Axes and pylab_examples example code: annotation_demo.py.

3.5 Image tutorial

3.5.1 Startup commands

In this tutorial we will work interactively with images. To do so we will use the IPython shell. You can start
it with:

$ipython

At the very least, you’ll need to have access to the imshow() function. The easy way for an interactive
environment: is to use the matplotlib API (see Artist tutorial) where you use explicit namespaces and control
object creation, etc...:
.. sourcecode:: ipython

    In [1]: import matplotlib.pyplot as plt
    In [2]: import matplotlib.image as mpimg
    In [3]: import numpy as np

You can now access functions like \texttt{imshow()} by using: \texttt{plt.imshow(yourimage)}. You can learn more about these functions in the \textit{Pyplot tutorial}.

### 3.5.2 Importing image data into Numpy arrays

Plotting image data is supported by the Pillow). Natively, matplotlib only supports PNG images. The commands shown below fall back on Pillow if the native read fails.

The image used in this example is a PNG file, but keep that Pillow requirement in mind for your own data. Here’s the image we’re going to play with:

![Image](image.png)

It’s a 24-bit RGB PNG image (8 bits for each of R, G, B). Depending on where you get your data, the other kinds of image that you’ll most likely encounter are RGBA images, which allow for transparency, or single-channel grayscale (luminosity) images. You can right click on it and choose “Save image as” to download it to your computer for the rest of this tutorial.

And here we go...
In [4]: img = mpimg.imread('stinkbug.png')
Out[4]:
array([[ 0.40784314,  0.40784314,  0.40784314],
        [ 0.40784314,  0.40784314,  0.40784314],
        [ 0.40784314,  0.40784314,  0.40784314],
        ...
        [ 0.42745098,  0.42745098,  0.42745098],
        [ 0.42745098,  0.42745098,  0.42745098],
        [ 0.42745098,  0.42745098,  0.42745098]],
       [[ 0.41176471,  0.41176471,  0.41176471],
        [ 0.41176471,  0.41176471,  0.41176471],
        [ 0.41176471,  0.41176471,  0.41176471],
        ...
        [ 0.42745098,  0.42745098,  0.42745098],
        [ 0.42745098,  0.42745098,  0.42745098],
        [ 0.42745098,  0.42745098,  0.42745098]],
       [[ 0.41960785,  0.41960785,  0.41960785],
        [ 0.41568628,  0.41568628,  0.41568628],
        [ 0.41568628,  0.41568628,  0.41568628],
        ...
        [ 0.43137255,  0.43137255,  0.43137255],
        [ 0.43137255,  0.43137255,  0.43137255],
        [ 0.43137255,  0.43137255,  0.43137255]],
       ...
       ...
       ...
       ...
       [[ 0.44313726,  0.44313726,  0.44313726],
        [ 0.44313726,  0.44313726,  0.44313726],
        [ 0.43921569,  0.43921569,  0.43921569],
        ...
        [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
        [ 0.4509804 ,  0.4509804 ,  0.4509804 ]],
       [[ 0.44313726,  0.44313726,  0.44313726],
        [ 0.44313726,  0.44313726,  0.44313726],
        [ 0.43921569,  0.43921569,  0.43921569],
        ...
        [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
        [ 0.44705883,  0.44705883,  0.44705883]],
       [[ 0.44313726,  0.44313726,  0.44313726],
        [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
        [ 0.4509804 ,  0.4509804 ,  0.4509804 ]],
       [[ 0.44313726,  0.44313726,  0.44313726],
        [ 0.44705883,  0.44705883,  0.44705883]],
       [[ 0.44313726,  0.44313726,  0.44313726]],
       dtype=float32)

Note the dtype there - float32. Matplotlib has rescaled the 8 bit data from each channel to floating point data between 0.0 and 1.0. As a side note, the only datatype that Pillow can work with is uint8. Matplotlib
plotting can handle float32 and uint8, but image reading/writing for any format other than PNG is limited to uint8 data. Why 8 bits? Most displays can only render 8 bits per channel worth of color gradation. Why can they only render 8 bits/channel? Because that’s about all the human eye can see. More here (from a photography standpoint): Luminous Landscape bit depth tutorial.

Each inner list represents a pixel. Here, with an RGB image, there are 3 values. Since it’s a black and white image, R, G, and B are all similar. An RGBA (where A is alpha, or transparency), has 4 values per inner list, and a simple luminance image just has one value (and is thus only a 2-D array, not a 3-D array). For RGB and RGBA images, matplotlib supports float32 and uint8 data types. For grayscale, matplotlib supports only float32. If your array data does not meet one of these descriptions, you need to rescale it.

3.5.3 Plotting numpy arrays as images

So, you have your data in a numpy array (either by importing it, or by generating it). Let’s render it. In Matplotlib, this is performed using the `imshow()` function. Here we’ll grab the plot object. This object gives you an easy way to manipulate the plot from the prompt.

```
In [5]: imgplot = plt.imshow(img)
```

You can also plot any numpy array - just remember that the datatype must be float32 (and range from 0.0 to 1.0) or uint8.
Applying pseudocolor schemes to image plots

Pseudocolor can be a useful tool for enhancing contrast and visualizing your data more easily. This is especially useful when making presentations of your data using projectors - their contrast is typically quite poor.

Pseudocolor is only relevant to single-channel, grayscale, luminosity images. We currently have an RGB image. Since R, G, and B are all similar (see for yourself above or in your data), we can just pick one channel of our data:

```
In [6]: lum_img = img[:,:,0]
```

This is array slicing. You can read more in the Numpy tutorial.

```
In [7]: imgplot = plt.imshow(lum_img)
```

Now, with a luminosity image, the default colormap (aka lookup table, LUT), is applied. The default is called jet. There are plenty of others to choose from. Let’s set some others using the `set_cmap()` method on our image plot object:
In [8]: imgplot.set_cmap('hot')

In [9]: imgplot.set_cmap('spectral')
There are many other colormap schemes available. See the list and images of the colormaps.

**Color scale reference**

It’s helpful to have an idea of what value a color represents. We can do that by adding color bars. It’s as easy as one line:

**In [10]:** plt.colorbar()
This adds a colorbar to your existing figure. This won’t automatically change if you change you switch to a different colormap - you have to re-create your plot, and add in the colorbar again.

**Examining a specific data range**

Sometimes you want to enhance the contrast in your image, or expand the contrast in a particular region while sacrificing the detail in colors that don’t vary much, or don’t matter. A good tool to find interesting regions is the histogram. To create a histogram of our image data, we use the `hist()` function.

```
In[10]: plt.hist(lum_img.flatten(), 256, range=(0.0,1.0), fc='k', ec='k')
```
Most often, the “interesting” part of the image is around the peak, and you can get extra contrast by clipping the regions above and/or below the peak. In our histogram, it looks like there’s not much useful information in the high end (not many white things in the image). Let’s adjust the upper limit, so that we effectively “zoom in on” part of the histogram. We do this by calling the `set_clim()` method of the image plot object.

In[11]: imgplot.set_clim(0.0,0.7)
Array Interpolation schemes

Interpolation calculates what the color or value of a pixel “should” be, according to different mathematical schemes. One common place that this happens is when you resize an image. The number of pixels change, but you want the same information. Since pixels are discrete, there’s missing space. Interpolation is how you fill that space. This is why your images sometimes come out looking pixelated when you blow them up. The effect is more pronounced when the difference between the original image and the expanded image is greater. Let’s take our image and shrink it. We’re effectively discarding pixels, only keeping a select few. Now when we plot it, that data gets blown up to the size on your screen. The old pixels aren’t there anymore, and the computer has to draw in pixels to fill that space.

```python
In [8]: from PIL import Image
In [9]: img = Image.open('stinkbug.png')  # Open image as Pillow image object
In [10]: rsize = img.resize((img.size[0]/10, img.size[1]/10))  # Use Pillow to resize
In [11]: rsizeArr = np.asarray(rsize)  # Get array back
In [12]: imgplot = plt.imshow(rsizeArr)
```
Here we have the default interpolation, bilinear, since we did not give `imshow()` any interpolation argument.

Let’s try some others:

```python
In [10]: imgplot.set_interpolation('nearest')
```
In [10]: imgplot.set_interpolation('bicubic')
Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.

### 3.6 Legend guide

This legend guide is an extension of the documentation available at `legend()` - please ensure you are familiar with contents of that documentation before proceeding with this guide.

This guide makes use of some common terms, which are documented here for clarity:

- **legend entry** A legend is made up of one or more legend entries. An entry is made up of exactly one key and one label.
- **legend key** The colored/patterned marker to the left of each legend label.
- **legend label** The text which describes the handle represented by the key.
- **legend handle** The original object which is used to generate an appropriate entry in the legend.

#### 3.6.1 Controlling the legend entries

Calling `legend()` with no arguments automatically fetches the legend handles and their associated labels. This functionality is equivalent to:
Matplotlib, Release 1.4.3

handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)

The `get_legend_handles_labels()` function returns a list of handles/artists which exist on the Axes which can be used to generate entries for the resulting legend - it is worth noting however that not all artists can be added to a legend, at which point a “proxy” will have to be created (see `Creating artists specifically for adding to the legend (aka. Proxy artists)` for further details).

For full control of what is being added to the legend, it is common to pass the appropriate handles directly to `legend()`:

```python
line_up, = plt.plot([1,2,3], label='Line 2')
line_down, = plt.plot([3,2,1], label='Line 1')
plt.legend(handles=[line_up, line_down])
```

In some cases, it is not possible to set the label of the handle, so it is possible to pass through the list of labels to `legend()`:

```python
line_up, = plt.plot([1,2,3], label='Line 2')
line_down, = plt.plot([3,2,1], label='Line 1')
plt.legend([line_up, line_down], ['Line Up', 'Line Down'])
```

### 3.6.2 Creating artists specifically for adding to the legend (aka. Proxy artists)

Not all handles can be turned into legend entries automatically, so it is often necessary to create an artist which can. Legend handles don’t have to exists on the Figure or Axes in order to be used.

Suppose we wanted to create a legend which has an entry for some data which is represented by a red color:

```python
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

red_patch = mpatches.Patch(color='red', label='The red data')
plt.legend(handles=[red_patch])
plt.show()
```
There are many supported legend handles, instead of creating a patch of color we could have created a line with a marker:

```python
import matplotlib.lines as mlines
import matplotlib.pyplot as plt

blue_line = mlines.Line2D([], [], color='blue', marker='*',
                          markersize=15, label='Blue stars')
plt.legend(handles=[blue_line])
plt.show()
```
3.6.3 Legend location

The location of the legend can be specified by the keyword argument `loc`. Please see the documentation at `legend()` for more details.

The `bbox_to_anchor` keyword gives a great degree of control for manual legend placement. For example, if you want your axes legend located at the figure’s top right-hand corner instead of the axes’ corner, simply specify the corner’s location, and the coordinate system of that location:

```python
plt.legend(bbox_to_anchor=(1, 1),
           bbox_transform=plt.gcf().transFigure)
```

More examples of custom legend placement:

```python
import matplotlib.pyplot as plt

plt.subplot(211)
plt.plot([1,2,3], label="test1")
plt.plot([3,2,1], label="test2")
# Place a legend above this legend, expanding itself to
# fully use the given bounding box.
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)
```
3.6.4 Multiple legends on the same Axes

Sometimes it is more clear to split legend entries across multiple legends. Whilst the instinctive approach to doing this might be to call the `legend()` function multiple times, you will find that only one legend ever exists on the Axes. This has been done so that it is possible to call `legend()` repeatedly to update the legend to the latest handles on the Axes, so to persist old legend instances, we must add them manually to the Axes:

```python
import matplotlib.pyplot as plt

line1, = plt.plot([1,2,3], label="Line 1", linestyle='--')
line2, = plt.plot([3,2,1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = plt.legend(handles=[line1], loc=1)
```
# Add the legend manually to the current Axes.
ax = plt.gca().add_artist(first_legend)

# Create another legend for the second line.
plt.legend(handles=[line2], loc=4)

plt.show()

3.6.5 Legend Handlers

In order to create legend entries, handles are given as an argument to an appropriate HandlerBase subclass. The choice of handler subclass is determined by the following rules:

1. Update get_legend_handler_map() with the value in the handler_map keyword.
2. Check if the handle is in the newly created handler_map.
3. Check if the type of handle is in the newly created handler_map.
4. Check if any of the types in the handle’s mro is in the newly created handler_map.

For completeness, this logic is mostly implemented in get_legend_handler().

All of this flexibility means that we have the necessary hooks to implement custom handlers for our own type of legend key.
The simplest example of using custom handlers is to instantiate one of the existing `HandlerBase` subclasses. For the sake of simplicity, let’s choose `matplotlib.legend_handler.HandlerLine2D` which accepts a `numpoints` argument (note `numpoints` is a keyword on the `legend()` function for convenience). We can then pass the mapping of instance to Handler as a keyword to `legend`.

```python
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D

line1, = plt.plot([3,2,1], marker='o', label='Line 1')
line2, = plt.plot([1,2,3], marker='o', label='Line 2')

plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})
```

As you can see, “Line 1” now has 4 marker points, where “Line 2” has 2 (the default). Try the above code, only change the map’s key from `line1` to `type(line1)`. Notice how now both `Line2D` instances get 4 markers.

Along with handlers for complex plot types such as errorbars, stem plots and histograms, the default `handler_map` has a special `tuple` handler (`HandlerTuple`) which simply plots the handles on top of one another for each item in the given tuple. The following example demonstrates combining two legend keys on top of one another:

```python
import matplotlib.pyplot as plt
from numpy.random import randn

3.6. Legend guide 81
z = randn(10)

red_dot, = plt.plot(z, "ro", markersize=15)
# Put a white cross over some of the data.
white_cross, = plt.plot(z[:5], "w+", markeredgewidth=3, markersize=15)

plt.legend([red_dot, (red_dot, white_cross)], ["Attr A", "Attr A+B"])

Implementing a custom legend handler

A custom handler can be implemented to turn any handle into a legend key (handles don’t necessarily need to be matplotlib artists). The handler must implement a “legend_artist” method which returns a single artist for the legend to use. Signature details about the “legend_artist” are documented at legend_artist().

```python
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

class AnyObject(object):
    pass

class AnyObjectHandler(object):
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
```
patch = mpatches.Rectangle([x0, y0], width, height, facecolor='red',
                        edgecolor='black', hatch='xx', lw=3,
                        transform=handlebox.get_transform())
handlebox.add_artist(patch)
return patch

plt.legend([AnyObject()], ['My first handler'],
            handler_map={AnyObject: AnyObjectHandler()})

Alternatively, had we wanted to globally accept AnyObject instances without needing to manually set the handler_map keyword all the time, we could have registered the new handler with:

```python
from matplotlib.legend import Legend
Legend.update_default_handler_map({AnyObject: AnyObjectHandler()})
```

Whilst the power here is clear, remember that there are already many handlers implemented and what you want to achieve may already be easily possible with existing classes. For example, to produce elliptical legend keys, rather than rectangular ones:

```python
from matplotlib.legend_handler import HandlerPatch
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

class HandlerEllipse(HandlerPatch):
```
def create_artists(self, legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans):
    center = 0.5 * width - 0.5 * xdescent, 0.5 * height - 0.5 * ydescent
    p = mpatches.Ellipse(xy=center, width=width + xdescent, height=height + ydescent)
    self.update_prop(p, orig_handle, legend)
    p.set_transform(trans)
    return [p]

c = mpatches.Circle((0.5, 0.5), 0.25, facecolor="green",
    edgecolor="red", linewidth=3)
plt.gca().add_patch(c)

plt.legend([c], ["An ellipse, not a rectangle"],
         handler_map={mpatches.Circle: HandlerEllipse()})

3.6.6 Known examples of using legend

Here is a non-exhaustive list of the examples available involving legend being used in various ways:

- lines_bars_and_markers example code: scatter_with_legend.py
- api example code: legend_demo.py
3.7 Annotating Axes

Do not proceed unless you already have read *Annotating text*, `text()` and `annotate()`!

3.7.1 Annotating with Text with Box

Let’s start with a simple example.

The `text()` function in the pyplot module (or text method of the Axes class) takes bbox keyword argument, and when given, a box around the text is drawn.
bbox_props = dict(boxstyle="rarrow,pad=0.3", fc="cyan", ec="b", lw=2)
t = ax.text(0, 0, "Direction", ha="center", va="center", rotation=45,
          size=15,
          bbox=bbox_props)

The patch object associated with the text can be accessed by:

    bb = t.get_bbox_patch()

The return value is an instance of FancyBboxPatch and the patch properties like facecolor, edgewidth, etc. can be accessed and modified as usual. To change the shape of the box, use set_boxstyle method.

    bb.set_boxstyle("rarrow", pad=0.6)

The arguments are the name of the box style with its attributes as keyword arguments. Currently, following box styles are implemented:

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RAxrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>
Note that the attributes arguments can be specified within the style name with separating comma (this form can be used as “boxstyle” value of bbox argument when initializing the text instance)

```python
bb.set_boxstyle("rarrow,pad=0.6")
```

### 3.7.2 Annotating with Arrow

The `annotate()` function in the pyplot module (or annotate method of the Axes class) is used to draw an arrow connecting two points on the plot.

```python
ax.annotate("Annotation",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='offset points',
            )
```

This annotates a point at `xy` in the given coordinate (`xycoords`) with the text at `xytext` given in `textcoords`. Often, the annotated point is specified in the `data` coordinate and the annotating text in `offset`
points. See `annotate()` for available coordinate systems.

An arrow connecting two point (xy & xytext) can be optionally drawn by specifying the `arrowprops` argument. To draw only an arrow, use empty string as the first argument.

```python
ax.annotate("",
    xy=(0.2, 0.2), xycoords='data',
    xytext=(0.8, 0.8), textcoords='data',
    arrowprops=dict(arrowstyle="->",
                    connectionstyle="arc3"),
)
```

The arrow drawing takes a few steps.

1. a connecting path between two points are created. This is controlled by `connectionstyle` key value.
2. If patch object is given (`patchA & patchB`), the path is clipped to avoid the patch.
3. The path is further shrunk by given amount of pixels (`shrinkA & shrinkB`)
4. The path is transmuted to arrow patch, which is controlled by the `arrowstyle` key value.
The creation of the connecting path between two points is controlled by connectionstyle key and following styles are available.

<table>
<thead>
<tr>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>angleA=90,angleB=0,rad=0.0</td>
</tr>
<tr>
<td>angle3</td>
<td>angleA=90,angleB=0</td>
</tr>
<tr>
<td>arc</td>
<td>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</td>
</tr>
<tr>
<td>arc3</td>
<td>rad=0.0</td>
</tr>
<tr>
<td>bar</td>
<td>armA=0.0,armB=0.0,fraction=0.3,angle=None</td>
</tr>
</tbody>
</table>

Note that “3” in angle3 and arc3 is meant to indicate that the resulting path is a quadratic spline segment (three control points). As will be discussed below, some arrow style option only can be used when the connecting path is a quadratic spline.

The behavior of each connection style is (limitedly) demonstrated in the example below. (Warning: The behavior of the bar style is currently not well defined, it may be changed in the future).

The connecting path (after clipping and shrinking) is then mutated to an arrow patch, according to the given arrowstyle.
Some arrow styles only work with connection styles that generate a quadratic-spline segment. They are `fancy`, `simple`, and `wedge`. For these arrow styles, you must use “angle3” or “arc3” connection style.

If the annotation string is given, the patchA is set to the bbox patch of the text by default.
As in the text command, a box around the text can be drawn using the bbox argument.

By default, the starting point is set to the center of the text extent. This can be adjusted with relpos key value. The values are normalized to the extent of the text. For example, (0,0) means lower-left corner and (1,1) means top-right.
3.7.3 Placing Artist at the anchored location of the Axes

There are class of artist that can be placed at the anchored location of the Axes. A common example is the legend. This type of artists can be created by using the OffsetBox class. A few predefined classes are available in `mpl_toolkits.axes_grid.anchored_artists`.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredText

at = AnchoredText("Figure 1a",
  prop=dict(size=8), frameon=True,
  loc=2,
)
at.set_boxstyle("round,pad=0.,rounding_size=0.2")
ax.add_artist(at)
```

The `loc` keyword has same meaning as in the legend command.
A simple application is when the size of the artist (or collection of artists) is known in pixel size during the time of creation. For example, if you want to draw a circle with fixed size of 20 pixel x 20 pixel (radius = 10 pixel), you can utilize AnchoredDrawingArea. The instance is created with a size of the drawing area (in pixel). And user can add arbitrary artist to the drawing area. Note that the extents of the artists that are added to the drawing area has nothing to do with the placement of the drawing area itself. The initial size only matters.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredDrawingArea

ada = AnchoredDrawingArea(20, 20, 0, 0,
                           loc=1, pad=0., frameon=False)
p1 = Circle((10, 10), 10)
ada.drawing_area.add_artist(p1)
p2 = Circle((30, 10), 5, fc="r")
ada.drawing_area.add_artist(p2)
```

The artists that are added to the drawing area should not have transform set (they will be overridden) and the dimension of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixel and 5 pixel, respectively.

![Graphical representation of circles](image)

Sometimes, you want to your artists scale with data coordinate (or other coordinate than canvas pixel). You can use AnchoredAuxTransformBox class. This is similar to AnchoredDrawingArea except that the extent of the artist is determined during the drawing time respecting the specified transform.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredAuxTransformBox

box = AnchoredAuxTransformBox(ax.transData, loc=2)
el = Ellipse((0,0), width=0.1, height=0.4, angle=30)  # in data coordinates!
box.drawing_area.add_artist(el)
```

The ellipse in the above example will have width and height corresponds to 0.1 and 0.4 in data coordinate and will be automatically scaled when the view limits of the axes change.

3.7. Annotating Axes 93
As in the legend, the bbox_to_anchor argument can be set. Using the HPacker and VPack, you can have an arrangement(?) of artist as in the legend (as a matter of fact, this is how the legend is created).

Note that unlike the legend, the bbox_transform is set to IdentityTransform by default.

### 3.7.4 Using Complex Coordinate with Annotation

The Annotation in matplotlib support several types of coordinate as described in *Annotating text*. For an advanced user who wants more control, it supports a few other options.

1. **Transform** instance. For example,
   ```python
   ax.annotate("Test", xy=(0.5, 0.5), xycoords=ax.transAxes)
   ```
   is identical to
ax.annotate("Test", xy=(0.5, 0.5), xycoords="axes fraction")

With this, you can annotate a point in other axes.

ax1, ax2 = subplot(121), subplot(122)
ax2.annotate("Test", xy=(0.5, 0.5), xycoords=ax1.transData,
    xytext=(0.5, 0.5), textcoords=ax2.transData,
    arrowprops=dict(arrowstyle="->"))

2. **Artist** instance. The xy value (or xytext) is interpreted as a fractional coordinate of the bbox (return value of `get_window_extent`) of the artist.

    an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                    va="center", ha="center",
                    bbox=dict(boxstyle="round", fc="w"))

    an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1, # (1,0.5) of the an1's bbox
                      xytext=(30,0), textcoords="offset points",
                      va="center", ha="left",
                      bbox=dict(boxstyle="round", fc="w"),
                      arrowprops=dict(arrowstyle="->"))

Note that it is your responsibility that the extent of the coordinate artist (an1 in above example) is determined before an2 gets drawn. In most cases, it means that an2 needs to be drawn later than an1.

3. A callable object that returns an instance of either BboxBase or Transform. If a transform is returned, it is same as 1 and if bbox is returned, it is same as 2. The callable object should take a single argument of renderer instance. For example, following two commands give identical results

    an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1,
                      xytext=(30,0), textcoords="offset points")

    an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1.get_window_extent,
                      xytext=(30,0), textcoords="offset points")

4. A tuple of two coordinate specification. The first item is for x-coordinate and the second is for y-coordinate. For example,

    annotate("Test", xy=(0.5, 1), xycoords=("data", "axes fraction"))

0.5 is in data coordinate, and 1 is in normalized axes coordinate. You may use an artist or transform as with a tuple. For example,
import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
    va="center", ha="center",
    bbox=dict(boxstyle="round", fc="w"))

an2 = ax.annotate("Test 2", xy=(0.5, 1.), xycoords=an1,
    xytext=(0.5,1.1), textcoords=(an1, "axes fraction"),
    va="bottom", ha="center",
    bbox=dict(boxstyle="round", fc="w"),
    arrowprops=dict(arrowstyle="->"))

plt.show()

5. Sometimes, you want your annotation with some “offset points”, but not from the annotated point but from other point. OffsetFrom is a helper class for such case.

import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
    va="center", ha="center",
    bbox=dict(boxstyle="round", fc="w"))

an2 = ax.annotate("Test 2", xy=(0.1, 0.1), xycoords="data",
    xytext=(0.5,1.1), textcoords=an1,
    # xytext is offset points from "xy=(0.5, 0), xycoords=an1"
    va="top", ha="center",
    bbox=dict(boxstyle="round", fc="w"),
    arrowprops=dict(arrowstyle="->"))

plt.show()}
You may take a look at this example `pylab_examples example code: annotation_demo3.py`.

### 3.7.5 Using ConnectorPatch

The ConnectorPatch is like an annotation without a text. While the `annotate` function is recommended in most of situation, the ConnectorPatch is useful when you want to connect points in different axes.

```python
from matplotlib.patches import ConnectionPatch
xy = (0.2, 0.2)
con = ConnectionPatch(xyA=xy, xyB=xy, coordsA="data", coordsB="data",
                           axesA=ax1, axesB=ax2)
ax2.add_artist(con)
```

The above code connects point xy in data coordinate of ax1 to point xy int data coordinate of ax2. Here is a simple example.

While the ConnectorPatch instance can be added to any axes, but you may want it to be added to the axes in the latter (?) of the axes drawing order to prevent overlap (?) by other axes.
Advanced Topics

3.7.6 Zoom effect between Axes

mpl_toolkits.axes_grid.inset_locator defines some patch classes useful for interconnect two axes. Understanding the code requires some knowledge of how mpl’s transform works. But, utilizing it will be straightforward.

3.7.7 Define Custom BoxStyle

You can use a custom box style. The value for the boxstyle can be a callable object in following forms:

```python
def __call__(self, x0, y0, width, height, mutation_size, aspect_ratio=1.):
    
    """
    Given the location and size of the box, return the path of the box around it.

    - *x0*, *y0*, *width*, *height* : location and size of the box
    - *mutation_size* : a reference scale for the mutation.
    - *aspect_ratio* : aspect-ratio for the mutation.
```
Here is a complete example.

```python
path = ...
return path
```

However, it is recommended that you derive from the matplotlib.patches.BoxStyle._Base as demonstrated below.

```python
from matplotlib.path import Path
from matplotlib.patches import BoxStyle
import matplotlib.pyplot as plt

# we may derive from matplotlib.patches.BoxStyle._Base class.
# You need to override transmute method in this case.

class MyStyle(BoxStyle._Base):
    
    A simple box.

    def __init__(self, pad=0.3):
        
        The arguments need to be floating numbers and need to have default values.

        *pad*
        amount of padding

        self.pad = pad
        super(MyStyle, self).__init__()

    def transmute(self, x0, y0, width, height, mutation_size):
```

3.7. Annotating Axes 99
Given the location and size of the box, return the path of the box around it.

- *x0*, *y0*, *width*, *height*: location and size of the box
- *mutation_size*: a reference scale for the mutation.

Often, the *mutation_size* is the font size of the text. You don't need to worry about the rotation as it is automatically taken care of.

```python
# padding
pad = mutation_size * self.pad

# width and height with padding added.
width, height = width + 2.*pad, \n    height + 2.*pad,

# boundary of the padded box
x0, y0 = x0-pad, y0-pad,
x1, y1 = x0+width, y0 + height

cp = [(x0, y0),
    (x1, y0), (x1, y1), (x0, y1),
    (x0-pad, (y0+y1)/2.), (x0, y0),
    (x0, y0)]

com = [Path.MOVETO,
    Path.LINETO, Path.LINETO, Path.LINETO,
    Path.LINETO, Path.LINETO,
    Path.CLOSEPOLY]

path = Path(cp, com)

return path
```

# register the custom style
BoxStyle._style_list["angled"] = MyStyle

```python
plt.figure(1, figsize=(3,3))
ax = plt.subplot(111)
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,
    bbox=dict(boxstyle="angled,pad=0.5", alpha=0.2))

del BoxStyle._style_list["angled"]
```

plt.show()
Similarly, you can define custom ConnectionStyle and custom ArrowStyle. See the source code of lib/matplotlib/patches.py and check how each style class is defined.

### 3.8 Screenshots

Here you’ll find a host of example plots with the code that generated them.

#### 3.8.1 Simple Plot

Here’s a very basic `plot()` with text labels:
3.8.2 Subplot demo

Multiple axes (i.e. subplots) are created with the `subplot()` command:
A tale of 2 subplots

3.8.3 Histograms

The `hist()` command automatically generates histograms and returns the bin counts or probabilities:
3.8.4 Path demo

You can add arbitrary paths in matplotlib using the `matplotlib.path` module:
3.8.5 mplot3d

The mplot3d toolkit (see mplot3d tutorial and mplot3d Examples) has support for simple 3d graphs including surface, wireframe, scatter, and bar charts.
Thanks to John Porter, Jonathon Taylor, Reinier Heeres, and Ben Root for the mplot3d toolkit. This toolkit is included with all standard matplotlib installs.

### 3.8.6 Streamplot

The streamplot() function plots the streamlines of a vector field. In addition to simply plotting the streamlines, it allows you to map the colors and/or line widths of streamlines to a separate parameter, such as the speed or local intensity of the vector field.
3.8. Screenshots
This feature complements the `quiver()` function for plotting vector fields. Thanks to Tom Flannaghan and Tony Yu for adding the streamplot function.

3.8.7 Ellipses

In support of the Phoenix mission to Mars (which used matplotlib to display ground tracking of spacecraft), Michael Droettboom built on work by Charlie Moad to provide an extremely accurate 8-spline approximation to elliptical arcs (see `Arc`), which are insensitive to zoom level.
3.8.8 Bar charts

Bar charts are simple to create using the `bar()` command, which includes customizations such as error bars:
It’s also simple to create stacked bars (bar_stacked.py), candlestick bars (finance_demo.py), and horizontal bar charts (barh_demo.py).

### 3.8.9 Pie charts

The `pie()` command allows you to easily create pie charts. Optional features include auto-labeling the percentage of area, exploding one or more wedges from the center of the pie, and a shadow effect. Take a close look at the attached code, which generates this figure in just a few lines of code.
3.8.10 Table demo

The `table()` command adds a text table to an axes.
### 3.8.11 Scatter demo

The `scatter()` command makes a scatter plot with (optional) size and color arguments. This example plots changes in Google’s stock price, with marker sizes reflecting the trading volume and colors varying with time. Here, the alpha attribute is used to make semitransparent circle markers.
3.8.12 Slider demo

Matplotlib has basic GUI widgets that are independent of the graphical user interface you are using, allowing you to write cross GUI figures and widgets. See `matplotlib.widgets` and the widget examples.
3.8.13 Fill demo

The `fill()` command lets you plot filled curves and polygons:
Thanks to Andrew Straw for adding this function.

### 3.8.14 Date demo

You can plot date data with major and minor ticks and custom tick formatters for both.
See `matplotlib.ticker` and `matplotlib.dates` for details and usage.

### 3.8.15 Financial charts

You can make sophisticated financial plots by combining the various plot functions, layout commands, and labeling tools provided by matplotlib. The following example emulates one of the financial plots in ChartDirector:
3.8.16 Basemap demo

Jeff Whitaker’s Basemap add-on toolkit makes it possible to plot data on many different map projections. This example shows how to plot contours, markers and text on an orthographic projection, with NASA’s “blue marble” satellite image as a background.
Sorry, could not import Basemap

3.8.17 Log plots

The `semilogx()`, `semilogy()` and `loglog()` functions simplify the creation of logarithmic plots.
Thanks to Andrew Straw, Darren Dale and Gregory Lielens for contributions log-scaling infrastructure.

3.8.18 Polar plots

The polar() command generates polar plots.
3.8.19 Legends

The `legend()` command automatically generates figure legends, with MATLAB-compatible legend placement commands.
Thanks to Charles Twardy for input on the legend command.

3.8.20 Mathtext_examples

Below is a sampling of the many TeX expressions now supported by matplotlib’s internal mathtext engine. The mathtext module provides TeX style mathematical expressions using freetype2 and the BaKoMa computer modern or STIX fonts. See the matplotlib.mathtext module for additional details.
Matplotlib's math rendering engine

\[ W_{\delta_1 \rho_1 \sigma_2}^{3\beta} = U_{\delta_1 \rho_1}^{3\beta} + \frac{1}{8\pi^2} \int_{\alpha_2}^{\alpha_2'} d\alpha_2' \left[ \frac{U_{\delta_1 \rho_1}^{2\beta} - \alpha_2' U_{\rho_1 \sigma_2}^{1\beta}}{U_{\rho_1 \sigma_2}^{9\beta}} \right] \]

Subscripts and superscripts:
\[ \alpha_i > \beta_i, \alpha_{i+1}^j = \sin(2\pi f_j t_i) e^{-5t_i/\tau}, \ldots \]

Fractions, binomials and stacked numbers:
\[ \frac{3}{4}, \left( \frac{3}{4} \right), \frac{3}{4}, \left( \frac{5-\frac{1}{2}}{4} \right), \ldots \]

Radicals:
\[ \sqrt{2}, \sqrt[3]{x}, \ldots \]

Fonts:
Roman, Italic, Typewriter or CALLIGRAPHY

Accents:
\[ \hat{a}, \bar{a}, \breve{a}, \check{a}, \dot{a}, \ddot{a}, \vec{a}, \hat{xyz}, \bar{xyz}, \ldots \]

Greek, Hebrew:
\[ \alpha, \beta, \chi, \delta, \lambda, \mu, \Delta, \Gamma, \Omega, \Phi, \Pi, \Upsilon, \nabla, \Re, \Im, \tau, \lambda, \ldots \]

Delimiters, functions and Symbols:
\[ \prod, \int, \oint, \sum, \log, \sin, \approx, \oplus, \star, \alpha, \infty, \partial, \Re, \langle \rangle, \ldots \]

Matplotlib’s mathtext infrastructure is an independent implementation and does not require TeX or any external packages installed on your computer. See the tutorial at Writing mathematical expressions.

### 3.8.21 Native TeX rendering

Although matplotlib’s internal math rendering engine is quite powerful, sometimes you need TeX. Matplotlib supports external TeX rendering of strings with the `usetex` option.
TEX is Number \[ \sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}! \]

### 3.8.22 EEG demo

You can embed matplotlib into pygtk, wx, Tk, FLTK, or Qt applications. Here is a screenshot of an EEG viewer called pbrain, which is part of the NeuroImaging in Python suite NIPY.
The lower axes use \texttt{specgram()} to plot the spectrogram of one of the EEG channels.

For examples of how to embed matplotlib in different toolkits, see:

- \texttt{user\_interfaces} example code: \texttt{embedding\_in\_gtk2.py}
- \texttt{user\_interfaces} example code: \texttt{embedding\_in\_wx2.py}
- \texttt{user\_interfaces} example code: \texttt{mpl\_with\_glade.py}
- \texttt{user\_interfaces} example code: \texttt{embedding\_in\_qt4.py}
- \texttt{user\_interfaces} example code: \texttt{embedding\_in\_tk.py}

### 3.8.23 XKCD-style sketch plots

matplotlib supports plotting in the style of \texttt{xkcd}. 
THE DAY I REALIZED I COULD COOK BACON WHENEVER I WANTED

"Stove Ownership" from xkcd by Randall Monroe
3.9 Choosing Colormaps

3.9.1 Overview

The idea behind choosing a good colormap is to find a good representation in 3D colorspace for your data set. The best colormap for any given data set depends on many things including:

- Whether representing form or metric data ([Ware])
- Your knowledge of the data set (e.g., is there a critical value from which the other values deviate?)
- If there is an intuitive color scheme for the parameter you are plotting
- If there is a standard in the field the audience may be expecting

For many applications, a perceptual colormap is the best choice — one in which equal steps in data are perceived as equal steps in the color space. Researchers have found that the human brain perceives changes in the lightness parameter as changes in the data much better than, for example, changes in hue. Therefore, colormaps which have monotonically increasing lightness through the colormap will be better interpreted by the viewer.

Color can be represented in 3D space in various ways. One way to represent color is using CIELAB. In CIELAB, color space is represented by lightness, $L^*$; red-green, $a^*$; and yellow-blue, $b^*$. The lightness
parameter $L^*$ can then be used to learn more about how the matplotlib colormaps will be perceived by viewers.

An excellent starting resource for learning about human perception of colormaps is from [IBM].

### 3.9.2 Classes of colormaps

Colormaps are often split into several categories based on their function (see, e.g., [Moreland]):

1. Sequential: change in lightness and often saturation of color incrementally, often using a single hue; should be used for representing information that has ordering.
2. Diverging: change in lightness and possibly saturation of two different colors that meet in the middle at an unsaturated color; should be used when the information being plotted has a critical middle value, such as topography or when the data deviates around zero.
3. Qualitative: often are miscellaneous colors; should be used to represent information which does not have ordering or relationships.

### 3.9.3 Lightness of matplotlib colormaps

Here we examine the lightness values of the matplotlib colormaps. Note that some documentation on the colormaps is available ([list-colormaps]).

**Sequential**

For the Sequential plots, the lightness value increases monotonically through the colormaps. This is good. Some of the $L^*$ values in the colormaps span from 0 to 100 (binary and the other grayscale), and others start around $L^* = 20$. Those that have a smaller range of $L^*$ will accordingly have a smaller perceptual range. Note also that the $L^*$ function varies amongst the colormaps: some are approximately linear in $L^*$ and others are more curved.

**Sequential2**

Many of the $L^*$ values from the Sequential2 plots are monotonically increasing, but some (autumn, cool, spring, and winter) plateau or even go both up and down in $L^*$ space. Others (afmhot, copper, gist_heat, and hot) have kinks in the $L^*$ functions. Data that is being represented in a region of the colormap that is at a plateau or kink will lead to a perception of banding of the data in those values in the colormap (see [mycarta-banding] for an excellent example of this).

**Diverging**

For the Diverging maps, we want to have monotonically increasing $L^*$ values up to a maximum, which should be close to $L^* = 100$, followed by monotonically decreasing $L^*$ values. We are looking for approximately equal minimum $L^*$ values at opposite ends of the colormap. By these measures, BrBG and RdBu are good options. coolwarm is a good option, but it doesn’t span a wide range of $L^*$ values (see grayscale section below).
Qualitative

Qualitative colormaps are not aimed at being perceptual maps, but looking at the lightness parameter can verify that for us. The $L^*$ values move all over the place throughout the colormap, and are clearly not monotonically increasing. These would not be good options for use as perceptual colormaps.

Miscellaneous

Some of the miscellaneous colormaps have particular uses they have been created for. For example, gist_earth, ocean, and terrain all seem to be created for plotting topography (green/brown) and water depths (blue) together. We would expect to see a divergence in these colormaps, then, but multiple kinks may not be ideal, such as in gist_earth and terrain. CMRmap was created to convert well to grayscale, though it does appear to have some small kinks in $L^*$. cubehelix was created to vary smoothly in both lightness and hue, but appears to have a small hump in the green hue area.

The often-used jet colormap is included in this set of colormaps. We can see that the $L^*$ values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at [mycarta-jet].
Sequential colormaps

Lightness $L$
Sequential2 colormaps

afmhot  autumn  bone  cool  copper  gist_gray  gist_heat

gray  hot  pink  spring  summer  winter

Lightness $L^*$
Diverging colormaps
Qualitative colormaps

Matplotlib, Release 1.4.3

Chapter 3. Beginner's Guide
3.9.4 $L^*$ function

There are multiple approaches to finding the best function for $L^*$ across a colormap. Linear gives reasonable results (e.g., [mycarta-banding], [mycarta-lablinear]). However, the Weber-Fechner law, and more generally and recently, Stevens’ Law, indicates that a logarithmic or geometric relationship might be better (see effort on this front at [mycarta-cubelaw]).
Additional Parts Black

1
2
3
4

8
4
2
1
3.9.5 Grayscale conversion

Conversion to grayscale is important to pay attention to for printing publications that have color plots. If this is not paid attention to ahead of time, your readers may end up with indecipherable plots because the grayscale changes unpredictably through the colormap.

Conversion to grayscale is done in many different ways [bw]. Some of the better ones use a linear combination of the rgb values of a pixel, but weighted according to how we perceive color intensity. A nonlinear method of conversion to grayscale is to use the $L^*$ values of the pixels. In general, similar principles apply for this question as they do for presenting one’s information perceptually; that is, if a colormap is chosen that has monotonically increasing in $L^*$ values, it will print in a reasonable manner to grayscale.

With this in mind, we see that the Sequential colormaps have reasonable representations in grayscale. Some of the Sequential2 colormaps have decent enough grayscale representations, though some (autumn, spring, summer, winter) have very little grayscale change. If a colormap like this was used in a plot and then the plot was printed to grayscale, a lot of the information may map to the same gray values. The Diverging colormaps mostly vary from darker gray on the outer edges to white in the middle. Some (PuOr and seismic) have noticeably darker gray on one side than the other and therefore are not very symmetric. coolwarm has little range of gray scale and would print to a more uniform plot, losing a lot of detail. Note that overlaid, labeled contours could help differentiate between one side of the colormap vs. the other since color cannot be used once a plot is printed to grayscale. Many of the Qualitative and Miscellaneous colormaps, such as Accent, hsv, and jet, change from darker to lighter and back to darker gray throughout the colormap. This would make it impossible for a viewer to interpret the information in a plot once it is printed in grayscale.
Sequential colormaps

- binary
- Blues
- BuGn
- BuPu
gist_yarg
- GnBu
- Greens
- Greys
- Oranges
- OrRd
- PuBu
- PuBuGn
- PuRd
- Purples
- RdPu
- Reds
- YlGn
- YlGnBu
- YlOrBr
- YlOrRd
Sequential2 colormaps

- afmhot
- autumn
- bone
- cool
- copper
- gist_gray
- gist_heat
- gray
- hot
- pink
- spring
- summer
- winter

3.9. Choosing Colormaps
Diverging colormaps

- BrBG
- bwr
- coolwarm
- PiYG
- PRGn
- PuOr
- RdBu
- RdGy
- RdYlBu
- RdYlGn
- seismic
Qualitative colormaps

- Accent
- Dark2
- hsv
- Paired
- Pastel1
- Pastel2
- Set1
- Set2
- Set3
- spectral
3.9.6 Color vision deficiencies

There is a lot of information available about color blindness available (e.g., [colorblindness]). Additionally, there are tools available to convert images to how they look for different types of color vision deficiencies (e.g., [asp]).

The most common form of color vision deficiency involves differentiating between red and green. Thus, avoiding colormaps with both red and green will avoid many problems in general.

3.9.7 References
4.1 Artist tutorial

There are three layers to the matplotlib API. The `matplotlib.backend_bases.FigureCanvas` is the area onto which the figure is drawn, the `matplotlib.backend_bases.Renderer` is the object which knows how to draw on the `FigureCanvas`, and the `matplotlib.artist.Artist` is the object that knows how to use a renderer to paint onto the canvas. The `FigureCanvas` and `Renderer` handle all the details of talking to user interface toolkits like `wxPython` or drawing languages like PostScript®, and the `Artist` handles all the high level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of his time working with the `Artists`.

There are two types of `Artists`: primitives and containers. The primitives represent the standard graphical objects we want to paint onto our canvas: `Line2D`, `Rectangle`, `Text`, `AxesImage`, etc., and the containers are places to put them (`Axis`, `Axes` and `Figure`). The standard use is to create a `Figure` instance, use the `Figure` to create one or more `Axes` or `Subplot` instances, and use the `Axes` instance helper methods to create the primitives. In the example below, we create a `Figure` instance using `matplotlib.pyplot.figure()`, which is a convenience method for instantiating `Figure` instances and connecting them with your user interface or drawing toolkit `FigureCanvas`. As we will discuss below, this is not necessary – you can work directly with PostScript, PDF `Gtk+`, or `wxPython` `FigureCanvas` instances, instantiate your `Figures` directly and connect them yourselves – but since we are focusing here on the `Artist` API we’ll let `pyplot` handle some of those details for us:

```python
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2,1,1) # two rows, one column, first plot
```

The `Axes` is probably the most important class in the matplotlib API, and the one you will be working with most of the time. This is because the `Axes` is the plotting area into which most of the objects go, and the `Axes` has many special helper methods (`plot()`, `text()`, `hist()`, `imshow()`) to create the most common graphics primitives (`Line2D`, `Text`, `Rectangle`, `Image`, respectively). These helper methods will take your data (e.g., `numpy` arrays and strings) and create primitive `Artist` instances as needed (e.g., `Line2D`), add them to the relevant containers, and draw them when requested. Most of you are probably familiar with the `Subplot`, which is just a special case of an `Axes` that lives on a regular rows by columns grid of `Subplot` instances. If you want to create an `Axes` at an arbitrary location, simply use the `add_axes()` method which takes a list of `[left, bottom, width, height]` values in 0-1 relative figure coordinates:
fig2 = plt.figure()
ax2 = fig2.add_axes([0.15, 0.1, 0.7, 0.3])

Continuing with our example:

```python
import numpy as np
t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax.plot(t, s, color='blue', lw=2)
```

In this example, `ax` is the `Axes` instance created by the `fig.add_subplot` call above (remember `Subplot` is just a subclass of `Axes`) and when you call `ax.plot`, it creates a `Line2D` instance and adds it to the `Axes.lines` list. In the interactive `ipython` session below, you can see that the `Axes.lines` list is length one and contains the same line that was returned by the `line, = ax.plot...` call:

```python
In [101]: ax.lines[0]
Out[101]: <matplotlib.lines.Line2D instance at 0x19a95710>
In [102]: line
Out[102]: <matplotlib.lines.Line2D instance at 0x19a95710>
```

If you make subsequent calls to `ax.plot` (and the hold state is “on” which is the default) then additional lines will be added to the list. You can remove lines later simply by calling the list methods; either of these will work:

```python
del ax.lines[0]
ax.lines.remove(line)  # one or the other, not both!
```

The Axes also has helper methods to configure and decorate the x-axis and y-axis tick, tick labels and axis labels:

```python
xtext = ax.set_xlabel('my xdata')  # returns a Text instance
ytext = ax.set_ylabel('my ydata')
```

When you call `ax.set_xlabel`, it passes the information on the `Text` instance of the `XAxis`. Each `Axes` instance contains an `XAxis` and a `YAxis` instance, which handle the layout and drawing of the ticks, tick labels and axis labels.

Try creating the figure below.
4.1.1 Customizing your objects

Every element in the figure is represented by a matplotlib Artist, and each has an extensive list of properties to configure its appearance. The figure itself contains a Rectangle exactly the size of the figure, which you can use to set the background color and transparency of the figures. Likewise, each Axes bounding box (the standard white box with black edges in the typical matplotlib plot, has a Rectangle instance that determines the color, transparency, and other properties of the Axes. These instances are stored as member variables Figure.patch and Axes.patch (“Patch” is a name inherited from MATLAB, and is a 2D “patch” of color on the figure, e.g., rectangles, circles and polygons). Every matplotlib Artist has the following properties...
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>The transparency - a scalar from 0-1</td>
</tr>
<tr>
<td>animated</td>
<td>A boolean that is used to facilitate animated drawing</td>
</tr>
<tr>
<td>axes</td>
<td>The axes that the Artist lives in, possibly None</td>
</tr>
<tr>
<td>clip_box</td>
<td>The bounding box that clips the Artist</td>
</tr>
<tr>
<td>clip_on</td>
<td>Whether clipping is enabled</td>
</tr>
<tr>
<td>clip_path</td>
<td>The path the artist is clipped to</td>
</tr>
<tr>
<td>contains</td>
<td>A picking function to test whether the artist contains the pick point</td>
</tr>
<tr>
<td>figure</td>
<td>The figure instance the artist lives in, possibly None</td>
</tr>
<tr>
<td>label</td>
<td>A text label (e.g., for auto-labeling)</td>
</tr>
<tr>
<td>picker</td>
<td>A python object that controls object picking</td>
</tr>
<tr>
<td>transform</td>
<td>The transformation</td>
</tr>
<tr>
<td>visible</td>
<td>A boolean whether the artist should be drawn</td>
</tr>
<tr>
<td>zorder</td>
<td>A number which determines the drawing order</td>
</tr>
</tbody>
</table>

Each of the properties is accessed with an old-fashioned setter or getter (yes we know this irritates Pythonistas and we plan to support direct access via properties or traits but it hasn’t been done yet). For example, to multiply the current alpha by a half:

```python
a = o.get_alpha()
o.set_alpha(0.5*a)
```

If you want to set a number of properties at once, you can also use the `set` method with keyword arguments. For example:

```python
o.set(alpha=0.5, zorder=2)
```

If you are working interactively at the python shell, a handy way to inspect the Artist properties is to use the `matplotlib.artist.getp()` function (simply `getp()` in pylab), which lists the properties and their values. This works for classes derived from Artist as well, e.g., Figure and Rectangle. Here are the Figure rectangle properties mentioned above:

```python
In [149]: matplotlib.artist.getp(fig.patch)
   alpha = 1.0
   animated = False
   antialiased or aa = True
   axes = None
   clip_box = None
   clip_on = False
   clip_path = None
   contains = None
   edgecolor or ec = w
   facecolor or fc = 0.75
   figure = Figure(8.125x6.125)
   fill = 1
   hatch = None
   height = 1
   label =
   linewidth or lw = 1.0
   picker = None
   transform = <Affine object at 0x134cca84>
   verts = ((0, 0), (0, 1), (1, 1), (1, 0))
```
visible = True
width = 1
window_extent = <Bbox object at 0x134acbcc>
x = 0
y = 0
zorder = 1

The docstrings for all of the classes also contain the Artist properties, so you can consult the interactive “help” or the artists for a listing of properties for a given object.

4.1.2 Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to now how to get at that object. As mentioned in the introduction, there are two kinds of objects: primitives and containers. The primitives are usually the things you want to configure (the font of a Text instance, the width of a Line2D) although the containers also have some properties as well – for example the Axes Artist is a container that contains many of the primitives in your plot, but it also has properties like the xscale to control whether the xaxis is ‘linear’ or ‘log’. In this section we’ll review where the various container objects store the Artists that you want to get at.

4.1.3 Figure container

The top level container Artist is the matplotlib.figure.Figure, and it contains everything in the figure. The background of the figure is a Rectangle which is stored in Figure.patch. As you add subplots (add_subplot()) and axes (add_axes()) to the figure these will be appended to the Figure.axes. These are also returned by the methods that create them:

```
In [156]: fig = plt.figure()
In [157]: ax1 = fig.add_subplot(211)
In [158]: ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])
In [159]: ax1
Out[159]: <matplotlib.axes.Subplot instance at 0xd54b26c>
```

Because the figure maintains the concept of the “current axes” (see Figure.gca and Figure.sca) to support the pylab/pyplot state machine, you should not insert or remove axes directly from the axes list, but rather use the add_subplot() and add_axes() methods to insert, and the delaxes() method to delete. You are free however, to iterate over the list of axes or index into it to get access to Axes instances you want to customize. Here is an example which turns all the axes grids on:

```
for ax in fig.axes:
    ax.grid(True)
```
The figure also has its own text, lines, patches and images, which you can use to add primitives directly. The default coordinate system for the Figure will simply be in pixels (which is not usually what you want) but you can control this by setting the transform property of the Artist you are adding to the figure.

More useful is “figure coordinates” where (0, 0) is the bottom-left of the figure and (1, 1) is the top-right of the figure which you can obtain by setting the Artist transform to fig.transFigure:

In [191]: fig = plt.figure()

In [192]: l1 = matplotlib.lines.Line2D([0, 1], [0, 1],
                           transform=fig.transFigure, figure=fig)

In [193]: l2 = matplotlib.lines.Line2D([0, 1], [1, 0],
                           transform=fig.transFigure, figure=fig)

In [194]: fig.lines.extend([l1, l2])

In [195]: fig.canvas.draw()

Here is a summary of the Artists the figure contains
<table>
<thead>
<tr>
<th>Figure attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>A list of Axes instances (includes Subplot)</td>
</tr>
<tr>
<td>patch</td>
<td>The Rectangle background</td>
</tr>
<tr>
<td>images</td>
<td>A list of FigureImages patches - useful for raw pixel display</td>
</tr>
<tr>
<td>legends</td>
<td>A list of Figure Legend instances (different from Axes.legends)</td>
</tr>
<tr>
<td>lines</td>
<td>A list of Figure Line2D instances (rarely used, see Axes.lines)</td>
</tr>
<tr>
<td>patches</td>
<td>A list of Figure patches (rarely used, see Axes.patches)</td>
</tr>
<tr>
<td>texts</td>
<td>A list Figure Text instances</td>
</tr>
</tbody>
</table>

### 4.1.4 Axes container

The `matplotlib.axes.Axes` is the center of the matplotlib universe – it contains the vast majority of all the Artists used in a figure with many helper methods to create and add these Artists to itself, as well as helper methods to access and customize the Artists it contains. Like the `Figure`, it contains a `Patch` patch which is a `Rectangle` for Cartesian coordinates and a `Circle` for polar coordinates; this patch determines the shape, background and border of the plotting region:

```python
ax = fig.add_subplot(111)
rect = ax.patch  # a Rectangle instance
rect.set_facecolor('green')
```

When you call a plotting method, e.g., the canonical `plot()` and pass in arrays or lists of values, the method will create a `matplotlib.lines.Line2D()` instance, update the line with all the Line2D properties passed as keyword arguments, add the line to the `Axes.lines` container, and returns it to you:

```python
In [213]: x, y = np.random.rand(2, 100)
In [214]: line, = ax.plot(x, y, '- ', color='blue', linewidth=2)
```

`plot` returns a list of lines because you can pass in multiple x, y pairs to plot, and we are unpacking the first element of the length one list into the line variable. The line has been added to the `Axes.lines` list:

```python
In [229]: print ax.lines
[<matplotlib.lines.Line2D instance at 0xd378b0c>]
```

Similarly, methods that create patches, like `bar()` creates a list of rectangles, will add the patches to the `Axes.patches` list:

```python
In [233]: n, bins, rectangles = ax.hist(np.random.randn(1000), 50, facecolor='yellow')
In [234]: rectangles
Out[234]: <a list of 50 Patch objects>
In [235]: print len(ax.patches)
```

You should not add objects directly to the `Axes.lines` or `Axes.patches` lists unless you know exactly what you are doing, because the Axes needs to do a few things when it creates and adds an object. It sets the figure and axes property of the Artist, as well as the default Axes transformation (unless a transformation is set). It also inspects the data contained in the Artist to update the data structures controlling auto-scaling, so that the view limits can be adjusted to contain the plotted data. You can, nonetheless, create objects
yourself and add them directly to the Axes using helper methods like `add_line()` and `add_patch()`. Here is an annotated interactive session illustrating what is going on:

```
In [261]: fig = plt.figure()
In [262]: ax = fig.add_subplot(111)

# create a rectangle instance
In [263]: rect = matplotlib.patches.Rectangle( (1,1), width=5, height=12)

# by default the axes instance is None
In [264]: print rect.get_axes()
None

# and the transformation instance is set to the "identity transform"
In [265]: print rect.get_transform()
<Affine object at 0x13695544>

# now we add the Rectangle to the Axes
In [266]: ax.add_patch(rect)

# and notice that the ax.add_patch method has set the axes instance
In [267]: print rect.get_axes()
Axes(0.125,0.1;0.775x0.8)

# and the transformation has been set too
In [268]: print rect.get_transform()
<Affine object at 0x15009ca4>

# the default axes transformation is ax.transData
In [269]: print ax.transData
<Affine object at 0x15009ca4>

# notice that the xlims of the Axes have not been changed
In [270]: print ax.get_xlim()
(0.0, 1.0)

# but the data limits have been updated to encompass the rectangle
In [271]: print ax.dataLim.bounds
(1.0, 1.0, 5.0, 12.0)

# we can manually invoke the auto-scaling machinery
In [272]: ax.autoscale_view()

# and now the xlim are updated to encompass the rectangle
In [273]: print ax.get_xlim()
(1.0, 6.0)

# we have to manually force a figure draw
In [274]: ax.figure.canvas.draw()
```

There are many, many Axes helper methods for creating primitive Artists and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of Artist they create,
and where they store them

<table>
<thead>
<tr>
<th>Helper method</th>
<th>Artist</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>ax.annotate - text annotations</td>
<td>Annotate</td>
<td>ax.texts</td>
</tr>
<tr>
<td>ax.bar - bar charts</td>
<td>Rectangle</td>
<td>ax.patches</td>
</tr>
<tr>
<td>ax.errorbar - error bar plots</td>
<td>Line2D and Rectangle</td>
<td>ax.patches and ax.patches</td>
</tr>
<tr>
<td>ax.fill - shared area</td>
<td>Polygon</td>
<td>ax.patches</td>
</tr>
<tr>
<td>ax.hist - histograms</td>
<td>Rectangle</td>
<td>ax.patches</td>
</tr>
<tr>
<td>ax.imshow - image data</td>
<td>AxesImage</td>
<td>ax.patches</td>
</tr>
<tr>
<td>ax.legend - axes legends</td>
<td>Legend</td>
<td>ax.legend</td>
</tr>
<tr>
<td>ax.plot - xy plots</td>
<td>Line2D</td>
<td>ax.lines</td>
</tr>
<tr>
<td>ax.scatter - scatter charts</td>
<td>PolygonCollection</td>
<td>ax.collections</td>
</tr>
<tr>
<td>ax.text - text</td>
<td>Text</td>
<td>ax.texts</td>
</tr>
</tbody>
</table>

In addition to all of these Artists, the Axes contains two important Artist containers: the XAxis and YAxis, which handle the drawing of the ticks and labels. These are stored as instance variables xaxis and yaxis. The XAxis and YAxis containers will be detailed below, but note that the Axes contains many helper methods which forward calls on to the Axis instances so you often do not need to work with them directly unless you want to. For example, you can set the font size of the XAxis ticklabels using the Axes helper method:

```python
for label in ax.get_xticklabels():
    label.set_color('orange')
```

Below is a summary of the Artists that the Axes contains

<table>
<thead>
<tr>
<th>Axes attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>artists</td>
<td>A list of Artist instances</td>
</tr>
<tr>
<td>patch</td>
<td>Rectangle instance for Axes background</td>
</tr>
<tr>
<td>collections</td>
<td>A list of Collection instances</td>
</tr>
<tr>
<td>images</td>
<td>A list of AxesImage</td>
</tr>
<tr>
<td>legends</td>
<td>A list of Legend instances</td>
</tr>
<tr>
<td>lines</td>
<td>A list of Line2D instances</td>
</tr>
<tr>
<td>patches</td>
<td>A list of Patch instances</td>
</tr>
<tr>
<td>texts</td>
<td>A list of Text instances</td>
</tr>
<tr>
<td>xaxis</td>
<td>matplotlib.axis.XAxis instance</td>
</tr>
<tr>
<td>yaxis</td>
<td>matplotlib.axis.YAxis instance</td>
</tr>
</tbody>
</table>

4.1.5 Axis containers

The matplotlib.axis.Axis instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis. The Axis also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the Locator and Formatter instances which control where the ticks are placed and how they are represented as strings.

Each Axis object contains a label attribute (this is what pylab modifies in calls to xlabel() and ylabel()) as well as a list of major and minor ticks. The ticks areXTick and YTick instances, which contain the actual line and text primitives that render the ticks and ticklabels. Because the ticks are dynamically created as needed (e.g., when panning and zooming), you should access the lists of major and minor
ticks through their accessor methods `get_major_ticks()` and `get_minor_ticks()`. Although the ticks contain all the primitives and will be covered below, the `Axis` methods contain accessor methods to return the tick lines, tick labels, tick locations etc.:

```
In [285]: axis = ax.xaxis

In [286]: axis.get_ticklocs()
Out[286]: array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

In [287]: axis.get_ticklabels()
Out[287]: <a list of 10 Text major ticklabel objects>

# note there are twice as many ticklines as labels because by
# default there are tick lines at the top and bottom but only tick
# labels below the xaxis; this can be customized
In [288]: axis.get_ticklines()
Out[288]: <a list of 20 Line2D ticklines objects>

# by default you get the major ticks back
In [291]: axis.get_ticklines()
Out[291]: <a list of 20 Line2D ticklines objects>

# but you can also ask for the minor ticks
In [292]: axis.get_ticklines(minor=True)
Out[292]: <a list of 0 Line2D ticklines objects>
```

Here is a summary of some of the useful accessor methods of the `Axis` (these have corresponding setters where useful, such as `set_major_formatter`)

<table>
<thead>
<tr>
<th>Accessor method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_scale</code></td>
<td>The scale of the axis, e.g., ‘log’ or ‘linear’</td>
</tr>
<tr>
<td><code>get_view_interval</code></td>
<td>The interval instance of the axis view limits</td>
</tr>
<tr>
<td><code>get_data_interval</code></td>
<td>The interval instance of the axis data limits</td>
</tr>
<tr>
<td><code>get_gridlines</code></td>
<td>A list of grid lines for the <code>Axis</code></td>
</tr>
<tr>
<td><code>get_label</code></td>
<td>The axis label - a <code>Text</code> instance</td>
</tr>
<tr>
<td><code>get_ticklabels</code></td>
<td>A list of <code>Text</code> instances - keyword `minor=True</td>
</tr>
<tr>
<td><code>get_ticklines</code></td>
<td>A list of <code>Line2D</code> instances - keyword `minor=True</td>
</tr>
<tr>
<td><code>get_major_locator</code></td>
<td>The <code>matplotlib.ticker.Locator</code> instance for major ticks</td>
</tr>
<tr>
<td><code>get_major_formatter</code></td>
<td>The <code>matplotlib.ticker.Formatter</code> instance for major ticks</td>
</tr>
<tr>
<td><code>get_minor_locator</code></td>
<td>The <code>matplotlib.ticker.Locator</code> instance for minor ticks</td>
</tr>
<tr>
<td><code>get_minor_formatter</code></td>
<td>The <code>matplotlib.ticker.Formatter</code> instance for minor ticks</td>
</tr>
<tr>
<td><code>get_major_ticks</code></td>
<td>A list of <code>Tick</code> locations - keyword `minor=True</td>
</tr>
<tr>
<td><code>get_minor_ticks</code></td>
<td>A list of <code>Tick</code> instances for major ticks</td>
</tr>
<tr>
<td><code>get_major_ticks</code></td>
<td>A list of <code>Tick</code> instances for minor ticks</td>
</tr>
<tr>
<td><code>grid</code></td>
<td>Turn the grid on or off for the major or minor ticks</td>
</tr>
</tbody>
</table>

Here is an example, not recommended for its beauty, which customizes the axes and tick properties

```
import numpy as np
import matplotlib.pyplot as plt

# plt.figure creates a matplotlib.figure.Figure instance
```
fig = plt.figure()
rect = fig.patch  # a rectangle instance
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategrey')

for label in ax1.xaxis.get_ticklabels():
    # label is a Text instance
    label.set_color('red')
    label.set_rotation(45)
    label.set_fontsize(16)

for line in ax1.yaxis.get_ticklines():
    # line is a Line2D instance
    line.set_color('green')
    line.set_markersize(25)
    line.set_markeredgewidth(3)
4.1.6 Tick containers

The `matplotlib.axis.Tick` is the final container object in our descent from the `Figure` to the `Axes` to the `Axis` to the `Tick`. The Tick contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute of the Tick. In addition, there are boolean variables that determine whether the upper labels and ticks are on for the x-axis and whether the right labels and ticks are on for the y-axis.

<table>
<thead>
<tr>
<th>Tick attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick1line</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>tick2line</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>gridline</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>label1</td>
<td>Text instance</td>
</tr>
<tr>
<td>label2</td>
<td>Text instance</td>
</tr>
<tr>
<td>gridOn</td>
<td>boolean which determines whether to draw the tickline</td>
</tr>
<tr>
<td>tick1On</td>
<td>boolean which determines whether to draw the 1st tickline</td>
</tr>
<tr>
<td>tick2On</td>
<td>boolean which determines whether to draw the 2nd tickline</td>
</tr>
<tr>
<td>label1On</td>
<td>boolean which determines whether to draw tick label</td>
</tr>
<tr>
<td>label2On</td>
<td>boolean which determines whether to draw tick label</td>
</tr>
</tbody>
</table>

Here is an example which sets the formatter for the right side ticks with dollar signs and colors them green on the right side of the yaxis:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(100*np.random.rand(20))

formatter = ticker.FormatStrFormatter('$$%1.2f$$')
ax.yaxis.set_major_formatter(formatter)

for tick in ax.yaxis.get_major_ticks():
    tick.label1On = False
    tick.label2On = True
tick.label2.set_color('green')
```
4.2 Customizing Location of Subplot Using GridSpec

GridSpec specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

SubplotSpec specifies the location of the subplot in the given GridSpec.

subplot2grid a helper function that is similar to “pyplot.subplot” but uses 0-based indexing and let subplot to occupy multiple cells.

4.2.1 Basic Example of using subplot2grid

To use subplot2grid, you provide geometry of the grid and the location of the subplot in the grid. For a simple single-cell subplot:

ax = plt.subplot2grid((2,2),(0, 0))

is identical to

ax = plt.subplot(2,2,1)
Note that, unlike matplotlib’s subplot, the index starts from 0 in gridspec.

To create a subplot that spans multiple cells,

```python
ax2 = plt.subplot2grid((3,3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1, 2), rowspan=2)
```

For example, the following commands

```python
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (0,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1, 2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2, 0))
ax5 = plt.subplot2grid((3,3), (2, 1))
```

creates

![GridSpec and SubplotSpec](image.png)

### 4.2.2 GridSpec and SubplotSpec

You can create GridSpec explicitly and use them to create a Subplot.

For example,

```python
ax = plt.subplot2grid((2,2),(0, 0))
```

is equal to
import matplotlib.gridspec as gridspec

gs = gridspec.GridSpec(2, 2)
ax = plt.subplot(gs[0, 0])

A gridspec instance provides array-like (2d or 1d) indexing that returns the SubplotSpec instance. For, SubplotSpec that spans multiple cells, use slice.

ax2 = plt.subplot(gs[1,:-1])
ax3 = plt.subplot(gs[1:, -1])

The above example becomes

gs = gridspec.GridSpec(3, 3)
ax1 = plt.subplot(gs[0, :])
ax2 = plt.subplot(gs[1,:-1])
ax3 = plt.subplot(gs[1:, -1])
ax4 = plt.subplot(gs[-1,0])
ax5 = plt.subplot(gs[-1,-2])

4.2.3 Adjust GridSpec layout

When a GridSpec is explicitly used, you can adjust the layout parameters of subplots that are created from the gridspec.
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)

This is similar to `subplots_adjust`, but it only affects the subplots that are created from the given GridSpec.

The code below

```python
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
ax1 = plt.subplot(gs1[:-1, :])
ax2 = plt.subplot(gs1[-1, :-1])
ax3 = plt.subplot(gs1[-1, -1])

gs2 = gridspec.GridSpec(3, 3)
gs2.update(left=0.55, right=0.98, hspace=0.05)
ax4 = plt.subplot(gs2[:, :-1])
ax5 = plt.subplot(gs2[:-1, -1])
ax6 = plt.subplot(gs2[-1, -1])
```

creates

![GridSpec w/ different subplotpars](image)

### 4.2.4 GridSpec using SubplotSpec

You can create GridSpec from the SubplotSpec, in which case its layout parameters are set to that of the location of the given SubplotSpec.
gs0 = gridspec.GridSpec(1, 2)

gs00 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[0])
gs01 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[1])

4.2.5 A Complex Nested GridSpec using SubplotSpec

Here’s a more sophisticated example of nested gridspec where we put a box around each cell of the outer 4x4 grid, by hiding appropriate spines in each of the inner 3x3 grids.
4.2.6 GridSpec with Varying Cell Sizes

By default, GridSpec creates cells of equal sizes. You can adjust relative heights and widths of rows and columns. Note that absolute values are meaningless, only their relative ratios matter.

```python
gs = gridspec.GridSpec(2, 2,
    width_ratios=[1,2],
    height_ratios=[4,1]
)

ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
```
ax3 = plt.subplot(gs[2])
ax4 = plt.subplot(gs[3])

4.3 Tight Layout guide

tight_layout automatically adjusts subplot params so that the subplot(s) fits in to the figure area. This is an experimental feature and may not work for some cases. It only checks the extents of ticklabels, axis labels, and titles.

4.3.1 Simple Example

In matplotlib, the location of axes (including subplots) are specified in normalized figure coordinates. It can happen that your axis labels or titles (or sometimes even ticklabels) go outside the figure area, and are thus clipped.

plt.rcParams['savefig.facecolor'] = "0.8"

def example_plot(ax, fontsize=12):
    ax.plot([1, 2])
    ax.locator_params(nbins=3)
    ax.set_xlabel('x-label', fontsize=fontsize)
To prevent this, the location of axes needs to be adjusted. For subplots, this can be done by adjusting the subplot params (*Move the edge of an axes to make room for tick labels*). Matplotlib v1.1 introduces a new command `tight_layout()` that does this automatically for you.

```python
plt.tight_layout()
```
When you have multiple subplots, often you see labels of different axes overlapping each other.

```python
plt.close('all')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
```
tight_layout() will also adjust spacing between subplots to minimize the overlaps.

plt.tight_layout()
tight_layout() can take keyword arguments of pad, w_pad and h_pad. These control the extra padding around the figure border and between subplots. The pads are specified in fraction of fontsize.

plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
**tight_layout()** will work even if the sizes of subplots are different as far as their grid specification is compatible. In the example below, `ax1` and `ax2` are subplots of a 2x2 grid, while `ax3` is of a 1x2 grid.

```python
plt.close('all')
fig = plt.figure()
ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

plt.tight_layout()
```
It works with subplots created with `subplot2grid()`. In general, subplots created from the gridspec (`Customizing Location of Subplot Using GridSpec`) will work.

```python
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)

plt.tight_layout()
```
Although not thoroughly tested, it seems to work for subplots with aspect != “auto” (e.g., axes with images).

```python
arr = np.arange(100).reshape((10,10))
plt.close('all')
fig = plt.figure(figsize=(5,4))
ax = plt.subplot(111)
im = ax.imshow(arr, interpolation="none")
plt.tight_layout()
```
Caveats

- `tight_layout()` only considers ticklabels, axis labels, and titles. Thus, other artists may be clipped and also may overlap.

- It assumes that the extra space needed for ticklabels, axis labels, and titles is independent of original location of axes. This is often true, but there are rare cases where it is not.

- `pad=0` clips some of the texts by a few pixels. This may be a bug or a limitation of the current algorithm and it is not clear why it happens. Meanwhile, use of `pad` at least larger than 0.3 is recommended.

Use with GridSpec

GridSpec has its own `tight_layout()` method (the pyplot api `tight_layout()` also works).

```python
plt.close('all')
fig = plt.figure()

import matplotlib.gridspec as gridspec

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
```

4.3. Tight Layout guide
You may provide an optional rect parameter, which specifies the bounding box that the subplots will be fit inside. The coordinates must be in normalized figure coordinates and the default is (0, 0, 1, 1).

gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])
For example, this can be used for a figure with multiple gridspecs.

```python
gs2 = gridspec.GridSpec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title('')
    ax.set_xlabel('')

ax.set_xlabel('x-label', fontsize=12)

gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)
```
We may try to match the top and bottom of two grids

top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)

While this should be mostly good enough, adjusting top and bottom may require adjustment of hspace also. To update hspace & vspace, we call `tight_layout()` again with updated rect argument. Note that the rect argument specifies the area including the ticklabels, etc. Thus, we will increase the bottom (which is 0 for the normal case) by the difference between the `bottom` from above and the bottom of each gridspec. Same thing for the top.

top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.tight_layout(fig, rect=[None, 0 + (bottom-gs1.bottom), 0.5, 1 - (gs1.top-top)])
gs2.tight_layout(fig, rect=[0.5, 0 + (bottom-gs2.bottom), None, 1 - (gs2.top-top)], h_pad=0.5)
Use with AxesGrid1

While limited, the axes_grid1 toolkit is also supported.

```python
plt.close('all')
fig = plt.figure()

from mpl_toolkits.axes_grid1 import Grid
grid = Grid(fig, rect=111, nrows_ncols=(2, 2),
            axes_pad=0.25, label_mode='L',
            )

for ax in grid:
    example_plot(ax)
    ax.title.set_visible(False)

plt.tight_layout()
```

4.3. Tight Layout guide
Colorbar

If you create a colorbar with the `colorbar()` command, the created colorbar is an instance of Axes, not Subplot, so `tight_layout` does not work. With Matplotlib v1.1, you may create a colorbar as a subplot using the gridspec.

```python
plt.close('all')
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")
plt.colorbar(im, use_gridspec=True)
plt.tight_layout()
```
Another option is to use AxesGrid1 toolkit to explicitly create an axes for colorbar.

```python
plt.close('all')
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

from mpl_toolkits.axes_grid1 import make_axes_locatable
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", "5%%", pad="3%%")
plt.colorbar(im, cax=cax)
plt.tight_layout()
```
4.4 Event handling and picking

matplotlib works with a number of user interface toolkits (wxpython, tkinter, qt4, gtk, and macosx) and in order to support features like interactive panning and zooming of figures, it is helpful to the developers to have an API for interacting with the figure via key presses and mouse movements that is “GUI neutral” so we don’t have to repeat a lot of code across the different user interfaces. Although the event handling API is GUI neutral, it is based on the GTK model, which was the first user interface matplotlib supported. The events that are triggered are also a bit richer vis-a-vis matplotlib than standard GUI events, including information like which `matplotlib.axes.Axes` the event occurred in. The events also understand the matplotlib coordinate system, and report event locations in both pixel and data coordinates.

4.4.1 Event connections

To receive events, you need to write a callback function and then connect your function to the event manager, which is part of the `FigureCanvasBase`. Here is a simple example that prints the location of the mouse click and which button was pressed:

```python
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.random.rand(10))

def onclick(event):
    print 'button=%d, x=%d, y=%d, xdata=%f, ydata=%f'
    print (event.button, event.x, event.y, event.xdata, event.ydata)
```

```
cid = fig.canvas.mpl_connect('button_press_event', onclick)

The FigureCanvas method mpl_connect() returns a connection id which is simply an integer. When you want to disconnect the callback, just call:

fig.canvas.mpl_disconnect(cid)

**Note:** The canvas retains only weak references to the callbacks. Therefore if a callback is a method of a class instance, you need to retain a reference to that instance. Otherwise the instance will be garbage-collected and the callback will vanish.

Here are the events that you can connect to, the class instances that are sent back to you when the event occurs, and the event descriptions

<table>
<thead>
<tr>
<th>Event name</th>
<th>Class and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'button_press_event'</td>
<td>MouseEvent - mouse button is pressed</td>
</tr>
<tr>
<td>'button_release_event'</td>
<td>MouseEvent - mouse button is released</td>
</tr>
<tr>
<td>'draw_event'</td>
<td>DrawEvent - canvas draw</td>
</tr>
<tr>
<td>'key_press_event'</td>
<td>KeyEvent - key is pressed</td>
</tr>
<tr>
<td>'key_release_event'</td>
<td>KeyEvent - key is released</td>
</tr>
<tr>
<td>'motion_notify_event'</td>
<td>MouseEvent - mouse motion</td>
</tr>
<tr>
<td>'pick_event'</td>
<td>PickEvent - an object in the canvas is selected</td>
</tr>
<tr>
<td>'resize_event'</td>
<td>ResizeEvent - figure canvas is resized</td>
</tr>
<tr>
<td>'scroll_event'</td>
<td>MouseEvent - mouse scroll wheel is rolled</td>
</tr>
<tr>
<td>'figure_enter_event'</td>
<td>LocationEvent - mouse enters a new figure</td>
</tr>
<tr>
<td>'figure_leave_event'</td>
<td>LocationEvent - mouse leaves a figure</td>
</tr>
<tr>
<td>'axes_enter_event'</td>
<td>LocationEvent - mouse enters a new axes</td>
</tr>
<tr>
<td>'axes_leave_event'</td>
<td>LocationEvent - mouse leaves an axes</td>
</tr>
</tbody>
</table>

### 4.4.2 Event attributes

All matplotlib events inherit from the base class matplotlib.backend_bases.Event, which store the attributes:

- **name** the event name
- **canvas** the FigureCanvas instance generating the event
- **guiEvent** the GUI event that triggered the matplotlib event

The most common events that are the bread and butter of event handling are key press/release events and mouse press/release and movement events. The KeyEvent and MouseEvent classes that handle these events are both derived from the LocationEvent, which has the following attributes

- **x** x position - pixels from left of canvas
- **y** y position - pixels from bottom of canvas
- **inaxes** the Axes instance if mouse is over axes
- **xdata** x coord of mouse in data coords
Let’s look at a simple example of a canvas, where a simple line segment is created every time a mouse is pressed:

```python
from matplotlib import pyplot as plt

class LineBuilder:
    def __init__(self, line):
        self.line = line
        self.xs = list(line.get_xdata())
        self.ys = list(line.get_ydata())
        self.cid = line.figure.canvas.mpl_connect('button_press_event', self)

    def __call__(self, event):
        print 'click', event
        if event.inaxes!=self.line.axes: return
        self.xs.append(event.xdata)
        self.ys.append(event.ydata)
        self.line.set_data(self.xs, self.ys)
        self.line.figure.canvas.draw()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click to build line segments')
line, = ax.plot([0], [0])  # empty line
linebuilder = LineBuilder(line)

plt.show()
```

The `MouseEvent` that we just used is a `LocationEvent`, so we have access to the data and pixel coordinates in `event.x` and `event.xdata`. In addition to the `LocationEvent` attributes, it has

- **button**: button pressed None, 1, 2, 3, ‘up’, ‘down’ (up and down are used for scroll events)
- **key**: the key pressed: None, any character, ‘shift’, ‘win’, or ‘control’

### Draggable rectangle exercise

Write draggable rectangle class that is initialized with a `Rectangle` instance but will move its x,y location when dragged. Hint: you will need to store the original xy location of the rectangle which is stored as `rect.xy` and connect to the press, motion and release mouse events. When the mouse is pressed, check to see if the click occurs over your rectangle (see `matplotlib.patches.Rectangle.contains()`) and if it does, store the rectangle xy and the location of the mouse click in data coords. In the motion event callback, compute the deltax and deltay of the mouse movement, and add those deltas to the origin of the rectangle you stored. The redraw the figure. On the button release event, just reset all the button press data you stored as None.

Here is the solution:

```python
import numpy as np
import matplotlib.pyplot as plt
```
class DraggableRectangle:
    def __init__(self, rect):
        self.rect = rect
        self.press = None

    def connect(self):
        'connect to all the events we need'
        self.cidpress = self.rect.figure.canvas.mpl_connect('button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect('button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect('motion_notify_event', self.on_motion)

    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return
        contains, attrd = self.rect.contains(event)
        if not contains: return
        print 'event contains', self.rect.xy
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata

    def on_motion(self, event):
        'on motion we will move the rect if the mouse is over us'
        if self.press is None: return
        if event.inaxes != self.rect.axes: return
        x0, y0, xpress, ypress = self.press
        dx = event.xdata - xpress
        dy = event.ydata - ypress
        #print 'x0=%f, xpress=%f, event.xdata=%f, dx=%f, x0+dx=%f%(x0, xpress, event.xdata, dx, x0+dx)
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)

        self.rect.figure.canvas.draw()

    def on_release(self, event):
        'on release we reset the press data'
        self.press = None
        self.rect.figure.canvas.draw()

    def disconnect(self):
        'disconnect all the stored connection ids'
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []

4.4. Event handling and picking
```python
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()

Extra credit: use the animation blit techniques discussed in the animations recipe to make the animated drawing faster and smoother.

Extra credit solution:

# draggable rectangle with the animation blit techniques; see
# http://www.scipy.org/Cookbook/Matplotlib/Animations
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    lock = None # only one can be animated at a time
    def __init__(self, rect):
        self.rect = rect
        self.press = None
        self.background = None
    def connect(self):
        'connect to all the events we need'
        self.cidpress = self.rect.figure.canvas.mpl_connect('button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect('button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect('motion_notify_event', self.on_motion)
    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return
        if DraggableRectangle.lock is not None: return
        contains, attrd = self.rect.contains(event)
        if not contains: return
        print 'event contains', self.rect.xy
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata
        DraggableRectangle.lock = self

        # draw everything but the selected rectangle and store the pixel buffer
        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        self.rect.set_animated(True)
        canvas.draw()
        self.background = canvas.copy_from_bbox(self.rect.axes.bbox)

        # now redraw just the rectangle
        axes.draw_artist(self.rect)
```

# and blit just the redrawn area
canvas.blit(axes.bbox)

def on_motion(self, event):
    'on motion we will move the rect if the mouse is over us'
    if DraggableRectangle.lock is not self:
        return
    if event.inaxes != self.rect.axes: return
    x0, y0, xpress, ypress = self.press
    dx = event.xdata - xpress
    dy = event.ydata - ypress
    self.rect.set_x(x0 + dx)
    self.rect.set_y(y0 + dy)
    canvas = self.rect.figure.canvas
    axes = self.rect.axes
    # restore the background region
    canvas.restore_region(self.background)
    # redraw just the current rectangle
    axes.draw_artist(self.rect)
    # blit just the redrawn area
    canvas.blit(axes.bbox)

def on_release(self, event):
    'on release we reset the press data'
    if DraggableRectangle.lock is not self:
        return
    self.press = None
    DraggableRectangle.lock = None
    # turn off the rect animation property and reset the background
    self.rect.set_animated(False)
    self.background = None
    # redraw the full figure
    self.rect.figure.canvas.draw()

def disconnect(self):
    'disconnect all the stored connection ids'
    self.rect.figure.canvas.mpl_disconnect(self.cidpress)
    self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
    self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
Matplotlib, Release 1.4.3

drs.append(dr)
plt.show()

4.4.3 Mouse enter and leave

If you want to be notified when the mouse enters or leaves a figure or axes, you can connect to the figure/axes enter/leave events. Here is a simple example that changes the colors of the axes and figure background that the mouse is over:

"""
Illustrate the figure and axes enter and leave events by changing the frame colors on enter and leave
"""
import matplotlib.pyplot as plt
def enter_axes(event):
    print 'enter_axes', event.inaxes
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()

def leave_axes(event):
    print 'leave_axes', event.inaxes
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()

def enter_figure(event):
    print 'enter_figure', event.canvas.figure
    event.canvas.figure.patch.set_facecolor('red')
    event.canvas.draw()

def leave_figure(event):
    print 'leave_figure', event.canvas.figure
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1 = plt.figure()
fig1.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig1.add_subplot(211)
ax2 = fig1.add_subplot(212)
fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)

fig2 = plt.figure()
fig2.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig2.add_subplot(211)
ax2 = fig2.add_subplot(212)
fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()

4.4.4 Object picking

You can enable picking by setting the picker property of an Artist (e.g., a matplotlib Line2D, Text, Patch, Polygon, AxesImage, etc...)

There are a variety of meanings of the picker property:

- **None** picking is disabled for this artist (default)
- **boolean** if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- **float** if picker is a number it is interpreted as an epsilon tolerance in points and the the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event.
- **function** if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event. The signature is hit, props = picker(artist, mouseevent) to determine the hit test. If the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes

After you have enabled an artist for picking by setting the picker property, you need to connect to the figure canvas pick_event to get pick callbacks on mouse press events. e.g.:

```python
def pick_handler(event):
    mouseevent = event.mouseevent
    artist = event.artist
    # now do something with this...
```

The PickEvent which is passed to your callback is always fired with two attributes:

- **mouseevent the mouse event that generate the pick event. The** mouse event in turn has attributes like x and y (the coords in display space, e.g., pixels from left, bottom) and xdata, ydata (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which Axes the mouse is over, etc. See matplotlib.backend_bases.MouseEvent for details.

- **artist the Artist that generated the pick event.**

Additionally, certain artists like Line2D and PatchCollection may attach additional meta data like the indices into the data that meet the picker criteria (e.g., all the points in the line that are within the specified epsilon tolerance)
Simple picking example

In the example below, we set the line picker property to a scalar, so it represents a tolerance in points (72 points per inch). The onpick callback function will be called when the pick event it within the tolerance distance from the line, and has the indices of the data vertices that are within the pick distance tolerance. Our onpick callback function simply prints the data that are under the pick location. Different matplotlib Artists can attach different data to the PickEvent. For example, Line2D attaches the ind property, which are the indices into the line data under the pick point. See pick() for details on the PickEvent properties of the line. Here is the code:

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on points')
line, = ax.plot(np.random.rand(100), 'o', picker=5)  # 5 points tolerance

def onpick(event):
    thisline = event.artist
    xdata = thisline.get_xdata()
    ydata = thisline.get_ydata()
    ind = event.ind
    print 'onpick points:', zip(xdata[ind], ydata[ind])

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

Picking exercise

Create a data set of 100 arrays of 1000 Gaussian random numbers and compute the sample mean and standard deviation of each of them (hint: numpy arrays have a mean and std method) and make a xy marker plot of the 100 means vs the 100 standard deviations. Connect the line created by the plot command to the pick event, and plot the original time series of the data that generated the clicked on points. If more than one point is within the tolerance of the clicked on point, you can use multiple subplots to plot the multiple time series.

Exercise solution:

```python
"""
compute the mean and stddev of 100 data sets and plot mean vs stddev. When you click on one of the mu, sigma points, plot the raw data from the dataset that generated the mean and stddev
""

import numpy as np
import matplotlib.pyplot as plt

X = np.random.rand(100, 1000)
xs = np.mean(X, axis=1)
```
ys = np.std(X, axis=1)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance

def onpick(event):
    if event.artist!=line: return True

    N = len(event.ind)
    if not N: return True

    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N,1,subplotnum+1)
        ax.plot(X[dataind])
        ax.text(0.05, 0.9, 'mu=%1.3f
        sigma=%1.3f'(xs[dataind], ys[dataind]),
        transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()
    return True

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()

4.5 Transformations Tutorial

Like any graphics packages, matplotlib is built on top of a transformation framework to easily move between coordinate systems, the userland data coordinate system, the axes coordinate system, the figure coordinate system, and the display coordinate system. In 95% of your plotting, you won’t need to think about this, as it happens under the hood, but as you push the limits of custom figure generation, it helps to have an understanding of these objects so you can reuse the existing transformations matplotlib makes available to you, or create your own (see matplotlib.transforms). The table below summarizes the existing coordinate systems, the transformation object you should use to work in that coordinate system, and the description of that system. In the Transformation Object column, ax is a Axes instance, and fig is a Figure instance.
<table>
<thead>
<tr>
<th>Coordinate</th>
<th>Transformation Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data axes</td>
<td>ax.transData</td>
<td>The userland data coordinate system, controlled by the xlim and ylim axes.</td>
</tr>
<tr>
<td></td>
<td>ax.transAxes</td>
<td>The coordinate system of the Axes; (0,0) is bottom left of the axes, and (1,1) is top right of the axes.</td>
</tr>
<tr>
<td>figure</td>
<td>fig.transFigure</td>
<td>The coordinate system of the Figure; (0,0) is bottom left of the figure, and (1,1) is top right of the figure.</td>
</tr>
<tr>
<td>display</td>
<td>None</td>
<td>This is the pixel coordinate system of the display; (0,0) is the bottom left of the display, and (width, height) is the top right of the display in pixels. Alternatively, the identity transform (<code>matplotlib.transforms.IdentityTransform()</code>) may be used instead of None.</td>
</tr>
</tbody>
</table>

All of the transformation objects in the table above take inputs in their coordinate system, and transform the input to the display coordinate system. That is why the display coordinate system has None for the Transformation Object column – it already is in display coordinates. The transformations also know how to invert themselves, to go from display back to the native coordinate system. This is particularly useful when processing events from the user interface, which typically occur in display space, and you want to know where the mouse click or key-press occurred in your data coordinate system.

### 4.5.1 Data coordinates

Let’s start with the most commonly used coordinate, the data coordinate system. Whenever you add data to the axes, matplotlib updates the datalimits, most commonly updated with the `set_xlim()` and `set_ylim()` methods. For example, in the figure below, the data limits stretch from 0 to 10 on the x-axis, and -1 to 1 on the y-axis.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)
plt.show()
```
You can use the `ax.transData` instance to transform from your data to your display coordinate system, either a single point or a sequence of points as shown below:

```
In [14]: type(ax.transData)
Out[14]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [15]: ax.transData.transform((5, 0))
Out[15]: array([ 335.175,  247.0])

In [16]: ax.transData.transform(((5, 0), (1,2)))
Out[16]: array([[ 335.175,  247.0],
             [ 132.435,  642.2]])
```

You can use the `inverted()` method to create a transform which will take you from display to data coordinates:

```
In [41]: inv = ax.transData.inverted()

In [42]: type(inv)
Out[42]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [43]: inv.transform((335.175, 247.))
Out[43]: array([ 5.0,  0.0])
```

If your are typing along with this tutorial, the exact values of the display coordinates may differ if you have
a different window size or dpi setting. Likewise, in the figure below, the display labeled points are probably not the same as in the ipython session because the documentation figure size defaults are different.

![Figure showing data and display coordinates]

**Note:** If you run the source code in the example above in a GUI backend, you may also find that the two arrows for the data and display annotations do not point to exactly the same point. This is because the display point was computed before the figure was displayed, and the GUI backend may slightly resize the figure when it is created. The effect is more pronounced if you resize the figure yourself. This is one good reason why you rarely want to work in display space, but you can connect to the 'on_draw' Event to update figure coordinates on figure draws; see *Event handling and picking.*

When you change the x or y limits of your axes, the data limits are updated so the transformation yields a new display point. Note that when we just change the ylim, only the y-display coordinate is altered, and when we change the xlim too, both are altered. More on this later when we talk about the Bbox.

```
In [54]: ax.transData.transform((5, 0))
Out[54]: array([ 335.175, 247. ])

In [55]: ax.set_ylim(-1,2)
Out[55]: (-1, 2)

In [56]: ax.transData.transform((5, 0))
Out[56]: array([ 335.175, 181.13333333])

In [57]: ax.set_xlim(10,20)
```
4.5.2 Axes coordinates

After the data coordinate system, axes is probably the second most useful coordinate system. Here the point (0,0) is the bottom left of your axes or subplot, (0.5, 0.5) is the center, and (1.0, 1.0) is the top right. You can also refer to points outside the range, so (-0.1, 1.1) is to the left and above your axes. This coordinate system is extremely useful when placing text in your axes, because you often want a text bubble in a fixed, location, e.g., the upper left of the axes pane, and have that location remain fixed when you pan or zoom. Here is a simple example that creates four panels and labels them ‘A’, ‘B’, ‘C’, ‘D’ as you often see in journals.

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
for i, label in enumerate(('A', 'B', 'C', 'D')):
    ax = fig.add_subplot(2,2,i+1)
    ax.text(0.05, 0.95, label, transform=ax.transAxes,
            fontsize=16, fontweight='bold', va='top')

plt.show()
```
You can also make lines or patches in the axes coordinate system, but this is less useful in my experience than using `ax.transAxes` for placing text. Nonetheless, here is a silly example which plots some random dots in data space, and overlays a semi-transparent `Circle` centered in the middle of the axes with a radius one quarter of the axes – if your axes does not preserve aspect ratio (see `set_aspect()`), this will look like an ellipse. Use the pan/zoom tool to move around, or manually change the data xlim and ylim, and you will see the data move, but the circle will remain fixed because it is not in data coordinates and will always remain at the center of the axes.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

fig = plt.figure()
ax = fig.add_subplot(111)

x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y, 'go')  # plot some data in data coordinates

circ = patches.Circle((0.5, 0.5), 0.25, transform=ax.transAxes,
                        facecolor='yellow', alpha=0.5)

ax.add_patch(circ)

plt.show()
```
4.5.3 Blended transformations

Drawing in blended coordinate spaces which mix axes with data coordinates is extremely useful, for example to create a horizontal span which highlights some region of the y-data but spans across the x-axis regardless of the data limits, pan or zoom level, etc. In fact these blended lines and spans are so useful, we have built in functions to make them easy to plot (see axhline(), axvline(), axhspan(), axvspan()) but for didactic purposes we will implement the horizontal span here using a blended transformation. This trick only works for separable transformations, like you see in normal Cartesian coordinate systems, but not on inseparable transformations like the PolarTransform.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.random.randn(1000)

ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \ldots \sigma=2$', fontsize=16)

# the x coords of this transformation are data, and the
# y coord are axes
trans = transforms.blended_transform_factory(
    ax.transData, ax.transAxes)

# highlight the 1..2 stddev region with a span.
# We want x to be in data coordinates and y to
# span from 0..1 in axes coords
rect = patches.Rectangle((1,0), width=1, height=1,
    transform=trans, color='yellow',
    alpha=0.5)

ax.add_patch(rect)

plt.show()
4.5.4 Using offset transforms to create a shadow effect

One use of transformations is to create a new transformation that is offset from another transformation, e.g., to place one object shifted a bit relative to another object. Typically you want the shift to be in some physical dimension, like points or inches rather than in data coordinates, so that the shift effect is constant at different zoom levels and dpi settings.

One use for an offset is to create a shadow effect, where you draw one object identical to the first just to the right of it, and just below it, adjusting the zorder to make sure the shadow is drawn first and then the object it is shadowing above it. The transforms module has a helper transformation `ScaledTranslation`. It is instantiated with:

\[
\text{trans} = \text{ScaledTranslation}(\text{xt}, \text{yt}, \text{scale\_trans})
\]

where \(\text{xt}\) and \(\text{yt}\) are the translation offsets, and \(\text{scale\_trans}\) is a transformation which scales \(\text{xt}\) and \(\text{yt}\) at transformation time before applying the offsets. A typical use case is to use the figure \(\text{fig.dpi\_scale\_trans}\) transformation for the \(\text{scale\_trans}\) argument, to first scale \(\text{xt}\) and \(\text{yt}\) specified in points to display space before doing the final offset. The dpi and inches offset is a common-enough use case that we have a special helper function to create it in `matplotlib.transforms.offset_copy()`, which returns a new transform with an added offset. But in the example below, we'll create the offset transform ourselves. Note the use of the plus operator in:

\[
\text{offset} = \text{transforms.ScaledTranslation}(	ext{dx}, \text{dy}, \text{fig.dpi\_scale\_trans})
\]

\[
\text{shadow\_transform} = \text{ax.transData + offset}
\]

showing that can chain transformations using the addition operator. This code says: first apply the data transformation \(\text{ax.transData}\) and then translate the data by \(\text{dx}\) and \(\text{dy}\) points. In typography, a'point' is 1/72 inches, and by specifying your offsets in points, your figure will look the same regardless of the dpi resolution it is saved in.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

# make a simple sine wave
x = np.arange(0., 2., 0.01)
y = np.sin(2*np.pi*x)
line, = ax.plot(x, y, lw=3, color='blue')

# shift the object over 2 points, and down 2 points
dx, dy = 2/72., -2/72.
offset = transforms.ScaledTranslation(dx, dy, fig.dpi_scale_trans)
shadow_transform = ax.transData + offset

# now plot the same data with our offset transform;
# use the zorder to make sure we are below the line
```
ax.plot(x, y, lw=3, color='gray',
        transform=shadow_transform,
        zorder=0.5*line.get_zorder())

tax.set_title('creating a shadow effect with an offset transform')
plt.show()

4.5.5 The transformation pipeline

The ax.transData transform we have been working with in this tutorial is a composite of three different transformations that comprise the transformation pipeline from data -> display coordinates. Michael Droettboom implemented the transformations framework, taking care to provide a clean API that segregated the nonlinear projections and scales that happen in polar and logarithmic plots, from the linear affine transformations that happen when you pan and zoom. There is an efficiency here, because you can pan and zoom in your axes which affects the affine transformation, but you may not need to compute the potentially expensive nonlinear scales or projections on simple navigation events. It is also possible to multiply affine transformation matrices together, and then apply them to coordinates in one step. This is not true of all possible transformations.

Here is how the ax.transData instance is defined in the basic separable axis Axes class:

```python
self.transData = self.transScale + (self.transLimits + self.transAxes)
```
We’ve been introduced to the `transAxes` instance above in *Axes coordinates*, which maps the (0,0), (1,1) corners of the axes or subplot bounding box to display space, so let’s look at these other two pieces.

`self.transLimits` is the transformation that takes you from data to axes coordinates; i.e., it maps your view xlim and ylim to the unit space of the axes (and `transAxes` then takes that unit space to display space). We can see this in action here

```
In [80]: ax = subplot(111)
In [81]: ax.set_xlim(0, 10)
Out[81]: (0, 10)
In [82]: ax.set_ylim(-1,1)
Out[82]: (-1, 1)
In [84]: ax.transLimits.transform((0,-1))
Out[84]: array([ 0., 0.])
In [85]: ax.transLimits.transform((10,-1))
Out[85]: array([ 1., 0.])
In [86]: ax.transLimits.transform((10,1))
Out[86]: array([ 1., 1.])
In [87]: ax.transLimits.transform((5,0))
Out[87]: array([ 0.5, 0.5])
```

and we can use this same inverted transformation to go from the unit axes coordinates back to data coordinates.

```
In [90]: inv.transform((0.25, 0.25))
Out[90]: array([ 2.5, -0.5])
```

The final piece is the `self.transScale` attribute, which is responsible for the optional non-linear scaling of the data, e.g., for logarithmic axes. When an Axes is initially setup, this is just set to the identity transform, since the basic matplotlib axes has linear scale, but when you call a logarithmic scaling function like `semilogx()` or explicitly set the scale to logarithmic with `set_xscale()`, then the `ax.transScale` attribute is set to handle the nonlinear projection. The scales transforms are properties of the respective xaxis and yaxis Axis instances. For example, when you call `ax.set_xscale('log')`, the xaxis updates its scale to a `matplotlib.scale.LogScale` instance.

For non-separable axes the PolarAxes, there is one more piece to consider, the projection transformation. The `transData matplotlib.projections.polar.PolarAxes` is similar to that for the typical separable matplotlib Axes, with one additional piece `transProjection`:

```
scale. transData = scale. transScale + scale. transProjection + \n(scale. transProjectionAffine + scale. transAxes)
```

`transProjection` handles the projection from the space, e.g., latitude and longitude for map data, or radius and theta for polar data, to a separable Cartesian coordinate system. There are several projection examples in the `matplotlib.projections` package, and the best way to learn more is to open the source for those packages and see how to make your own, since matplotlib supports extensible axes and projections. Michael Droettboom has provided a nice tutorial example of creating a hammer projection axes; see *api example*
4.6 Path Tutorial

The object underlying all of the matplotlib.patch objects is the Path, which supports the standard set of moveto, lineto, curveto commands to draw simple and compound outlines consisting of line segments and splines. The Path is instantiated with a (N,2) array of (x,y) vertices, and a N-length array of path codes. For example to draw the unit rectangle from (0,0) to (1,1), we could use this code

```python
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

verts = [
    (0., 0.), # left, bottom
    (0., 1.), # left, top
    (1., 1.), # right, top
    (1., 0.), # right, bottom
    (0., 0.), # ignored
]

codes = [Path.MOVETO,
    Path.LINETO,
    Path.LINETO,
    Path.LINETO,
    Path.CLOSEPOLY,
]

path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='orange', lw=2)
ax.add_patch(patch)
ax.set_xlim(-2,2)
ax.set_ylim(-2,2)
plt.show()
```
The following path codes are recognized

<table>
<thead>
<tr>
<th>Code</th>
<th>Vertices</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>1 (ignored)</td>
<td>A marker for the end of the entire path (currently not required and ignored)</td>
</tr>
<tr>
<td>MOVETO</td>
<td>1</td>
<td>Pick up the pen and move to the given vertex.</td>
</tr>
<tr>
<td>LINETO</td>
<td>1</td>
<td>Draw a line from the current position to the given vertex.</td>
</tr>
<tr>
<td>CURVE3</td>
<td>2 (1 control point, 1 endpoint)</td>
<td>Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.</td>
</tr>
<tr>
<td>CURVE4</td>
<td>3 (2 control points, 1 endpoint)</td>
<td>Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.</td>
</tr>
<tr>
<td>CLOSEPOLY</td>
<td>(point itself is ignored)</td>
<td>Draw a line segment to the start point of the current polyline.</td>
</tr>
</tbody>
</table>

### 4.6.1 Bézier example

Some of the path components require multiple vertices to specify them: for example CURVE 3 is a Bézier curve with one control point and one end point, and CURVE4 has three vertices for the two control points and the end point. The example below shows a CURVE4 Bézier spline – the Bézier curve will be contained in the convex hull of the start point, the two control points, and the end point.

```python
import matplotlib.pyplot as plt
from matplotlib.path import Path
```
```python
import matplotlib.patches as patches

verts = [
    (0., 0.),  # P0
    (0.2, 1.),  # P1
    (1., 0.8),  # P2
    (0.8, 0.),  # P3
]

codes = [Path.MOVETO,
    Path.CURVE4,
    Path.CURVE4,
    Path.CURVE4,
    Path.CURVE4,
]

path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='none', lw=2)
ax.add_patch(patch)

xs, ys = zip(*verts)
ax.plot(xs, ys,
    'x--', lw=2, color='black', ms=10)

ax.text(-0.05, -0.05, 'P0')
ax.text(0.15, 1.05, 'P1')
ax.text(1.05, 0.85, 'P2')
ax.text(0.85, -0.05, 'P3')

ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
plt.show()
```
4.6.2 Compound paths

All of the simple patch primitives in matplotlib, Rectangle, Circle, Polygon, etc, are implemented with simple path. Plotting functions like `hist()` and `bar()`, which create a number of primitives, e.g., a bunch of Rectangles, can usually be implemented more efficiently using a compound path. The reason `bar()` creates a list of rectangles and not a compound path is largely historical: the `Path` code is comparatively new and `bar()` predates it. While we could change it now, it would break old code, so here we will cover how to create compound paths, replacing the functionality in `bar`, in case you need to do so in your own code for efficiency reasons, e.g., you are creating an animated bar plot.

We will make the histogram chart by creating a series of rectangles for each histogram bar: the rectangle width is the bin width and the rectangle height is the number of datapoints in that bin. First we'll create some random normally distributed data and compute the histogram. Because numpy returns the bin edges and not centers, the length of `bins` is 1 greater than the length of `n` in the example below:

```python
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)
```

We'll now extract the corners of the rectangles. Each of the `left`, `bottom`, etc, arrays below is `len(n)`, where `n` is the array of counts for each histogram bar:
# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n

Now we have to construct our compound path, which will consist of a series of MOVETO, LINETO and CLOSEPOLY for each rectangle. For each rectangle, we need 5 vertices: 1 for the MOVETO, 3 for the LINETO, and 1 for the CLOSEPOLY. As indicated in the table above, the vertex for the closepoly is ignored but we still need it to keep the codes aligned with the vertices:

nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5,0] = left
verts[0::5,1] = bottom
verts[1::5,0] = left
verts[1::5,1] = top
verts[2::5,0] = right
verts[2::5,1] = top
verts[3::5,0] = right
verts[3::5,1] = bottom

All that remains is to create the path, attach it to a PathPatch, and add it to our axes:

barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
                          edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

Here is the result
4.7 Path effects guide

Matplotlib’s `patheffects` module provides functionality to apply a multiple draw stage to any Artist which can be rendered via a `Path`.

Artists which can have a path effect applied to them include `Patch`, `Line2D`, `Collection` and even `Text`. Each artist’s path effects can be controlled via the `set_path_effects` method (`set_path_effects`), which takes an iterable of `AbstractPathEffect` instances.

The simplest path effect is the `Normal` effect, which simply draws the artist without any effect:

```python
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(5, 1.5))
text = fig.text(0.5, 0.5, 'Hello path effects world!
This is the normal 
path effect.
Pretty dull, huh?',
               ha='center', va='center', size=20)
text.set_path_effects([path_effects.Normal()])
plt.show()
```

Hello path effects world!
This is the normal path effect.
Pretty dull, huh?

Whilst the plot doesn’t look any different to what you would expect without any path effects, the drawing of the text now been changed to use the the path effects framework, opening up the possibilities for more interesting examples.

4.7.1 Adding a shadow

A far more interesting path effect than Normal is the drop-shadow, which we can apply to any of our path based artists. The classes SimplePatchShadow and SimpleLineShadow do precisely this by drawing either a filled patch or a line patch below the original artist:

```python
import matplotlib.pyplot as plt
import matplotlib.pathEffects as path_effects

text = plt.text(0.5, 0.5, 'Hello path effects world!',
                 path_effects=[path_effects.withSimplePatchShadow()])

plt.plot([0, 3, 2, 5], linewidth=5, color='blue',
          path_effects=[path_effects.SimpleLineShadow(),
                        path_effects.Normal()])

plt.show()
```
Notice the two approaches to setting the path effects in this example. The first uses the `with*` classes to include the desired functionality automatically followed with the “normal” effect, whereas the latter explicitly defines the two path effects to draw.

### 4.7.2 Making an artist stand out

One nice way of making artists visually stand out is to draw an outline in a bold color below the actual artist. The `Stroke` path effect makes this a relatively simple task:

```python
import matplotlib.pyplot as plt
import matplotlib.pathEffects as path_effects

fig = plt.figure(figsize=(7, 1))
text = fig.text(0.5, 0.5, 'This text stands out because of its black border.', color='white', ha='center', va='center', size=30)
text.set_path_effects([path_effects.Stroke(linewidth=3, foreground='black'), path_effects.Normal()])
plt.show()
```

**This text stands out because of its black border.**
It is important to note that this effect only works because we have drawn the text path twice; once with a thick black line, and then once with the original text path on top.

You may have noticed that the keywords to Stroke and SimplePatchShadow and SimpleLineShadow are not the usual Artist keywords (such as facecolor and edgecolor etc.). This is because with these path effects we are operating at lower level of matplotlib. In fact, the keywords which are accepted are those for a matplotlib.backend_bases.GraphicsContextBase instance, which have been designed for making it easy to create new backends - and not for its user interface.

### 4.7.3 Greater control of the path effect artist

As already mentioned, some of the path effects operate at a lower level than most users will be used to, meaning that setting keywords such as facecolor and edgecolor raise an AttributeError. Luckily there is a generic PathPatchEffect path effect which creates a PathPatch class with the original path. The keywords to this effect are identical to those of PathPatch:

```python
import matplotlib.pyplot as plt
import matplotlib.path as path
fig = plt.figure(figsize=(8, 1))
t = fig.text(0.02, 0.5, 'Hatch shadow', fontsize=75, weight=1000, va='center')
t.set_path_effects([path.PathPatchEffect(offset=(4, -4), hatch='xxxx', facecolor='gray'),
                     path.PathPatchEffect(edgecolor='white', linewidth=1.1, facecolor='black')])
plt.show()
```

### 4.8 Our Favorite Recipes

Here is a collection of short tutorials, examples and code snippets that illustrate some of the useful idioms and tricks to make snazzier figures and overcome some matplotlib warts.

#### 4.8.1 Sharing axis limits and views

It’s common to make two or more plots which share an axis, e.g., two subplots with time as a common axis. When you pan and zoom around on one, you want the other to move around with you. To facilitate this, matplotlib Axes support a sharex and sharey attribute. When you create a subplot() or axes() instance, you can pass in a keyword indicating what axes you want to share with

```python
In [96]: t = np.arange(0, 10, 0.01)
In [97]: ax1 = plt.subplot(211)
```
In [98]: ax1.plot(t, np.sin(2*np.pi*t))
Out[98]: [-matplotlib.lines.Line2D object at 0x98719ec]

In [99]: ax2 = plt.subplot(212, sharex=ax1)
In [100]: ax2.plot(t, np.sin(4*np.pi*t))
Out[100]: [-matplotlib.lines.Line2D object at 0xb7d8fec]

### 4.8.2 Easily creating subplots

In early versions of matplotlib, if you wanted to use the pythonic API and create a figure instance and from that create a grid of subplots, possibly with shared axes, it involved a fair amount of boilerplate code. e.g.

```python
# old style
fig = plt.figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(223, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(224, sharex=ax1, sharey=ax1)
```

Fernando Perez has provided a nice top level method to create in `subplots()` (note the “s” at the end) everything at once, and turn off x and y sharing for the whole bunch. You can either unpack the axes individually:

```python
# new style method 1; unpack the axes
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True, sharey=True)
ax1.plot(x)
```
or get them back as a numrows x numcolumns object array which supports numpy indexing:

```python
# new style method 2; use an axes array
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
axs[0,0].plot(x)
```

### 4.8.3 Fixing common date annoyances

matplotlib allows you to natively plots python datetime instances, and for the most part does a good job picking tick locations and string formats. There are a couple of things it does not handle so gracefully, and here are some tricks to help you work around them. We’ll load up some sample date data which contains datetime.date objects in a numpy record array:

```python
In [63]: datafile = cbook.get_sample_data('goog.npy')
In [64]: r = np.load(datafile).view(np.recarray)
```

```python
In [65]: r.dtype
Out[65]: dtype([('date', '<M8[D]'), ('open', '<f8'), ('high', '<f8'), ('low', '<f8'), ('close', '<f8'), ('volume', '<i8'), ('adj_close', '<f8'))]
```

4.8. Our Favorite Recipes 203
In [66]: r.date
Out[66]:
array([2004-08-19, 2004-08-20, 2004-08-23, ..., 2008-10-10, 2008-10-13,
       2008-10-14], dtype=object)

The dtype of the numpy record array for the field date is |O4 which means it is a 4-byte python object pointer; in this case the objects are datetime.date instances, which we can see when we print some samples in the ipython terminal window.

If you plot the data,

In [67]: plot(r.date, r.close)
Out[67]: [<matplotlib.lines.Line2D object at 0x92a6b6c>]

you will see that the x tick labels are all squashed together.

```
Dec 2004
Jun 2005
Dec 2005
Jun 2006
Dec 2006
Jun 2007
Dec 2007
Jun 2008
100
200
300
400
500
600
700
800
```

Another annoyance is that if you hover the mouse over the window and look in the lower right corner of the matplotlib toolbar (Interactive navigation) at the x and y coordinates, you see that the x locations are formatted the same way the tick labels are, e.g., “Dec 2004”. What we’d like is for the location in the toolbar to have a higher degree of precision, e.g., giving us the exact date out mouse is hovering over. To fix the first problem, we can use `matplotlib.figure.Figure.autofmt_xdate()` and to fix the second problem we can use the `ax.fmt_xdata` attribute which can be set to any function that takes a scalar and returns a string. matplotlib has a number of date formatters built in, so we’ll use one of those.
plt.close('all')
fig, ax = plt.subplots(1)
ax.plot(r.date, r.close)

# rotate and align the tick labels so they look better
fig.autofmt_xdate()

# use a more precise date string for the x axis locations in the
# toolbar
import matplotlib.dates as mdates
ax.fmt_xdata = mdates.DateFormatter('%Y-%m-%d')
plt.title('fig.autofmt_xdate fixes the labels')

Now when you hover your mouse over the plotted data, you’ll see date format strings like 2004-12-01 in the toolbar.

### 4.8.4 Fill Between and Alpha

The `fill_between()` function generates a shaded region between a min and max boundary that is useful for illustrating ranges. It has a very handy `where` argument to combine filling with logical ranges, e.g., to just fill in a curve over some threshold value.

At its most basic level, `fill_between` can be use to enhance a graphs visual appearance. Let’s compare two graphs of a financial times with a simple line plot on the left and a filled line on the right.
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook

# load up some sample financial data
datafile = cbook.get_sample_data('goog.npy')
r = np.load(datafile).view(np.recarray)

# create two subplots with the shared x and y axes
fig, (ax1, ax2) = plt.subplots(1, 2, sharex=True, sharey=True)

pricemin = r.close.min()

ax1.plot(r.date, r.close, lw=2)
ax2.fill_between(r.date, pricemin, r.close, facecolor='blue', alpha=0.5)

for ax in ax1, ax2:
    ax.grid(True)

ax1.set_ylabel('price')
for label in ax2.get_yticklabels():
    label.set_visible(False)

fig.suptitle('Google (GOOG) daily closing price')
fig.autofmt_xdate()
The alpha channel is not necessary here, but it can be used to soften colors for more visually appealing plots. In other examples, as we’ll see below, the alpha channel is functionally useful as the shaded regions can overlap and alpha allows you to see both. Note that the postscript format does not support alpha (this is a postscript limitation, not a matplotlib limitation), so when using alpha save your figures in PNG, PDF or SVG.

Our next example computes two populations of random walkers with a different mean and standard deviation of the normal distributions from which the steps are drawn. We use shared regions to plot +/- one standard deviation of the mean position of the population. Here the alpha channel is useful, not just aesthetic.

```python
import matplotlib.pyplot as plt
import numpy as np

Nsteps, Nwalkers = 100, 250
t = np.arange(Nsteps)

# an (Nsteps x Nwalkers) array of random walk steps
S1 = 0.002 + 0.01*np.random.randn(Nsteps, Nwalkers)
S2 = 0.004 + 0.02*np.random.randn(Nsteps, Nwalkers)

# an (Nsteps x Nwalkers) array of random walker positions
X1 = S1.cumsum(axis=0)
X2 = S2.cumsum(axis=0)
```
# Nsteps length arrays empirical means and standard deviations of both
# populations over time
mu1 = X1.mean(axis=1)
sigma1 = X1.std(axis=1)
mu2 = X2.mean(axis=1)
sigma2 = X2.std(axis=1)

# plot it!
fig, ax = plt.subplots(1)
ax.plot(t, mu1, lw=2, label='mean population 1', color='blue')
ax.plot(t, mu1, lw=2, label='mean population 2', color='yellow')
ax.fill_between(t, mu1+sigma1, mu1-sigma1, facecolor='blue', alpha=0.5)
ax.fill_between(t, mu2+sigma2, mu2-sigma2, facecolor='yellow', alpha=0.5)
ax.set_title('random walkers empirical $\mu$ and $\pm \sigma$ interval')
ax.legend(loc='upper left')
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
shade that region blue.

```python
np.random.seed(1234)

Nsteps = 500
t = np.arange(Nsteps)

mu = 0.002
sigma = 0.01

# the steps and position
S = mu + sigma*np.random.randn(Nsteps)
X = S.cumsum()

# the 1 sigma upper and lower analytic population bounds
lower_bound = mu*t - sigma*np.sqrt(t)
upper_bound = mu*t + sigma*np.sqrt(t)

fig, ax = plt.subplots(1)
ax.plot(t, X, lw=2, label='walker position', color='blue')
ax.plot(t, mu*t, lw=1, label='population mean', color='black', ls='--')
ax.fill_between(t, lower_bound, upper_bound, facecolor='yellow', alpha=0.5,
                label='1 sigma range')
ax.legend(loc='upper left')

# here we use the where argument to only fill the region where the
# walker is above the population 1 sigma boundary
ax.fill_between(t, upper_bound, X, where=X>upper_bound, facecolor='blue', alpha=0.5)
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```
Another handy use of filled regions is to highlight horizontal or vertical spans of an axes – for that matplotlib has some helper functions `axhspan()` and `axvspan()` and example `pylab_examples example code: axhspan_demo.py`.

### 4.8.5 Transparent, fancy legends

Sometimes you know what your data looks like before you plot it, and may know for instance that there won’t be much data in the upper right hand corner. Then you can safely create a legend that doesn’t overlay your data:

```
ax.legend(loc='upper right')
```

Other times you don’t know where your data is, and loc='best' will try and place the legend:

```
ax.legend(loc='best')
```

but still, your legend may overlap your data, and in these cases it’s nice to make the legend frame transparent.

```
np.random.seed(1234)
fig, ax = plt.subplots(1)
ax.plot(np.random.randn(300), 'o-', label='normal distribution')
ax.plot(np.random.rand(300), 's-', label='uniform distribution')
ax.set_ylim(-3, 3)
ax.legend(loc='best', fancybox=True, framealpha=0.5)
```
4.8.6 Placing text boxes

When decorating axes with text boxes, two useful tricks are to place the text in axes coordinates (see Transformations Tutorial), so the text doesn’t move around with changes in x or y limits. You can also use the bbox property of text to surround the text with a Patch instance – the bbox keyword argument takes a dictionary with keys that are Patch properties.

```python
np.random.seed(1234)
fig, ax = plt.subplots(1)
x = 30*np.random.randn(10000)
mu = x.mean()
median = np.median(x)
sigma = x.std()
textstr = r'$\mu=%.2f$
\mathrm{median}=%.2f$\sigma=%.2f'$%(mu, median, sigma)
ax.hist(x, 50)
# these are matplotlib.patch.Patch properties
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)

# place a text box in upper left in axes coords
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
```
verticalalignment='top', bbox=props)

$\mu = 0.48$
$\text{median} = 0.54$
$\sigma = 29.86$
WHAT’S NEW IN MATPLOTLIB

This page just covers the highlights – for the full story, see the CHANGELOG

For a list of all of the issues and pull requests since the last revision, see the github-stats.

Note: matplotlib 1.4 supports Python 2.6, 2.7, 3.3, and 3.4
matplotlib 1.3 supports Python 2.6, 2.7, 3.2, and 3.3
matplotlib 1.2 supports Python 2.6, 2.7, and 3.1
matplotlib 1.1 supports Python 2.4 to 2.7
Table of Contents

• What’s new in matplotlib
  – new in matplotlib-1.4
    * New colormap
    * The nbagg backend
    * New plotting features
    * Date handling
    * Configuration (rcParams)
    * style package added
    * Backends
    * Text
    * Sphinx extensions
    * Legend and PathEffects documentation
    * Widgets
    * GAE integration
  – new in matplotlib-1.3
    * New in 1.3.1
    * New plotting features
    * Updated Axes3D.contour methods
    * Drawing
    * Text
    * Configuration (rcParams)
    * Backends
    * Documentation and examples
    * Infrastructure
  – new in matplotlib 1.2.2
    * Improved collections
    * Multiple images on same axes are correctly transparent
  – new in matplotlib-1.2
    * Python 3.x support
    * PGF/TikZ backend
    * Locator interface
    * Tri-Surface Plots
    * Control the lengths of colorbar extensions
    * Figures are picklable
    * Set default bounding box in matplotlibrc
    * New Boxplot Functionality
    * New RC parameter functionality
    * Streamplot
    * New hist functionality
    * Updated shipped dependencies
    * Face-centred colors in tripcolor plots
    * Hatching patterns in filled contour plots, with legends
    * Known issues in the matplotlib-1.2 release
  – new in matplotlib-1.1
    * Sankey Diagrams
    * Animation
    * Tight Layout
    * PyQt4, PySide, and IPython
    * Legend
    * mplot3d
    * Numerix support removed
    * Marker
5.1 new in matplotlib-1.4

Thomas A. Caswell served as the release manager for the 1.4 release.

5.1.1 New colormap

In heatmaps, a green-to-red spectrum is often used to indicate intensity of activity, but this can be problematic for the red/green colorblind. A new, colorblind-friendly colormap is now available at matplotlib.cm.Wistia. This colormap maintains the red/green symbolism while achieving deuteranopic legibility through brightness variations. See here for more information.

5.1.2 The nbagg backend

Phil Elson added a new backend, named “nbagg”, which enables interactive figures in a live IPython notebook session. The backend makes use of the infrastructure developed for the webagg backend, which itself gives standalone server backed interactive figures in the browser, however nbagg does not require a dedicated matplotlib server as all communications are handled through the IPython Comm machinery.

As with other backends nbagg can be enabled inside the IPython notebook with:

```python
import matplotlib
matplotlib.use('nbagg')
```

Once figures are created and then subsequently shown, they will placed in an interactive widget inside the notebook allowing panning and zooming in the same way as any other matplotlib backend. Because figures require a connection to the IPython notebook server for their interactivity, once the notebook is saved, each figure will be rendered as a static image - thus allowing non-interactive viewing of figures on services such as nbviewer.

5.1.3 New plotting features

Power-law normalization

Ben Gamari added a power-law normalization method, PowerNorm. This class maps a range of values to the interval [0,1] with power-law scaling with the exponent provided by the constructor’s gamma argument. Power law normalization can be useful for, e.g., emphasizing small populations in a histogram.

Fully customizable boxplots

Paul Hobson overhauled the boxplot() method such that it is now completely customizable in terms of the styles and positions of the individual artists. Under the hood, boxplot() relies on a new function (boxplot_stats()), which accepts any data structure currently compatible with boxplot(), and returns a list of dictionaries containing the positions for each element of the boxplots. Then a second method, bxp() is called to draw the boxplots based on the stats.
The `boxplot()` function can be used as before to generate boxplots from data in one step. But now the user has the flexibility to generate the statistics independently, or to modify the output of `boxplot_stats()` prior to plotting with `bxp()`.

Lastly, each artist (e.g., the box, outliers, cap, notches) can now be toggled on or off and their styles can be passed in through individual kwargs. See the examples: statistics example code: `boxplot_demo.py` and statistics example code: `bxp_demo.py`

Added a bool kwarg, `manage_xticks`, which if False disables the management of the ticks and limits on the x-axis by `bxp()`.

**Support for datetime axes in 2d plots**

Andrew Dawson added support for datetime axes to `contour()`, `contourf()`, `pcolormesh()` and `pcolor()`.

**Support for additional spectrum types**

Todd Jennings added support for new types of frequency spectrum plots: `magnitude_spectrum()`, `phase_spectrum()`, and `angle_spectrum()`, as well as corresponding functions in mlab.

He also added these spectrum types to `specgram()`, as well as adding support for linear scaling there (in addition to the existing dB scaling). Support for additional spectrum types was also added to `specgram()`. He also increased the performance for all of these functions and plot types.

**Support for detrending and windowing 2D arrays in mlab**

Todd Jennings added support for 2D arrays in the `detrend_mean()`, `detrend_none()`, and `detrend()`, as well as adding `apply_window()` which support windowing 2D arrays.

**Support for strides in mlab**

Todd Jennings added some functions to mlab to make it easier to use numpy strides to create memory-efficient 2D arrays. This includes `stride_repeat()`, which repeats an array to create a 2D array, and `stride_windows()`, which uses a moving window to create a 2D array from a 1D array.

**Formatter for new-style formatting strings**

Added `FormatStrFormatterNewStyle` which does the same job as `FormatStrFormatter`, but accepts new-style formatting strings instead of printf-style formatting strings.

**Consistent grid sizes in streamplots**

`streamplot()` uses a base grid size of 30x30 for both `density=1` and `density=(1, 1)`. Previously a grid size of 30x30 was used for `density=1`, but a grid size of 25x25 was used for `density=(1, 1)`. 
Get a list of all tick labels (major and minor)

Added the kwarg ‘which’ to get_xticklabels(), get_yticklabels() and get_ticklabels(). ‘which’ can be ‘major’, ‘minor’, or ‘both’ select which ticks to return, like set_ticks_position(). If ‘which’ is None then the old behaviour (controlled by the bool minor).

Separate horizontal/vertical axes padding support in ImageGrid

The kwarg ‘axes_pad’ to mpl_toolkits.axes_grid1.ImageGrid can now be a tuple if separate horizontal/vertical padding is needed. This is supposed to be very helpful when you have a labelled legend next to every subplot and you need to make some space for legend’s labels.

Support for skewed transformations

The Affine2D gained additional methods skew and skew_deg to create skewed transformations. Additionally, matplotlib internals were cleaned up to support using such transforms in Axes. This transform is important for some plot types, specifically the Skew-T used in meteorology.
Support for specifying properties of wedge and text in pie charts.

Added the kwargs ‘wedgeprops’ and ‘textprops’ to pie() to accept properties for wedge and text objects in a pie. For example, one can specify wedgeprops = {'linewidth':3} to specify the width of the borders of the wedges in the pie. For more properties that the user can specify, look at the docs for the wedge and text objects.

Fixed the direction of errorbar upper/lower limits

Larry Bradley fixed the errorbar() method such that the upper and lower limits (lolims, uplims, xlolims, xuplims) now point in the correct direction.
More consistent add-object API for Axes

Added the Axes method `add_image()` to put image handling on a par with artists, collections, containers, lines, patches, and tables.

Violin Plots

Per Parker, Gregory Kelsie, Adam Ortiz, Kevin Chan, Geoffrey Lee, Deokjae Donald Seo, and Taesu Terry Lim added a basic implementation for violin plots. Violin plots can be used to represent the distribution of sample data. They are similar to box plots, but use a kernel density estimation function to present a smooth approximation of the data sample used. The added features are:

`violin()` - Renders a violin plot from a collection of statistics. `violin_stats()` - Produces a collection of statistics suitable for rendering a violin plot. `violinplot()` - Creates a violin plot from a set of sample data. This method makes use of `violin_stats()` to process the input data, and `violin_stats()` to do the actual rendering. Users are also free to modify or replace the output of `violin_stats()` in order to customize the violin plots to their liking.

This feature was implemented for a software engineering course at the University of Toronto, Scarborough, run in Winter 2014 by Anya Tafliovich.

More markevery options to show only a subset of markers

Rohan Walker extended the `markevery` property in `Line2D`. You can now specify a subset of markers to show with an int, slice object, numpy fancy indexing, or float. Using a float shows markers at approximately equal display-coordinate-distances along the line.

Added size related functions to specialized Collections

Added the `get_size` and `set_size` functions to control the size of elements of specialized collections (`AsteriskPolygonCollection BrokenBarHCollection CircleCollection PathCollection PolyCollection RegularPolyCollection StarPolygonCollection`).

Fixed the mouse coordinates giving the wrong theta value in Polar graph

Added code to `transform_non_affine()` to ensure that the calculated theta value was between the range of 0 and 2 * pi since the problem was that the value can become negative after applying the direction and rotation to the theta calculation.

Simple quiver plot for mplot3d toolkit

A team of students in an Engineering Large Software Systems course, taught by Prof. Anya Tafliovich at the University of Toronto, implemented a simple version of a quiver plot in 3D space for the mplot3d toolkit as one of their term project. This feature is documented in `quiver()`. The team members are: Ryan Steve D’Souza, Victor B, xbtsw, Yang Wang, David, Caradec Bisesar and Vlad Vassilovski.
**polar-plot r-tick locations**

Added the ability to control the angular position of the r-tick labels on a polar plot via `set_rlabel_position()`.

### 5.1.4 Date handling

**n-d array support for date conversion**

Andrew Dawson added support for n-d array handling to `matplotlib.dates.num2date()`, `matplotlib.dates.date2num()` and `matplotlib.dates.datestr2num()`. Support is also added to the unit conversion interfaces `matplotlib.dates.DateConverter` and `matplotlib.units_REGISTRY`.

### 5.1.5 Configuration (rcParams)

**savefig.transparent added**

Controls whether figures are saved with a transparent background by default. Previously `savefig` always defaulted to a non-transparent background.
axes.titleweight

Added rcParam to control the weight of the title

axes.formatter.useoffset added

Controls the default value of useOffset in ScalarFormatter. If True and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. If False then the full number will be formatted in all conditions.

nbagg.transparent added

Controls whether nbagg figures have a transparent background. nbagg.transparent is True by default.

XDG compliance

Matplotlib now looks for configuration files (both rcparams and style) in XDG compliant locations.

5.1.6 style package added

You can now easily switch between different styles using the new style package:

```python
>>> from matplotlib import style
>>> style.use('dark_background')
```

Subsequent plots will use updated colors, sizes, etc. To list all available styles, use:

```python
>>> print style.available
```

You can add your own custom <style name>.mplstyle files to ~/.matplotlib/stylelib or call use with a URL pointing to a file with matplotlibrc settings.

*Note that this is an experimental feature*, and the interface may change as users test out this new feature.

5.1.7 Backends

Qt5 backend

Martin Fitzpatrick and Tom Badran implemented a Qt5 backend. The differences in namespace locations between Qt4 and Qt5 was dealt with by shimming Qt4 to look like Qt5, thus the Qt5 implementation is the primary implementation. Backwards compatibility for Qt4 is maintained by wrapping the Qt5 implementation.

The Qt5Agg backend currently does not work with IPython’s %matplotlib magic.

The 1.4.0 release has a known bug where the toolbar is broken. This can be fixed by:
Matplotlib, Release 1.4.3

cd path/to/installed/matplotlib
wget https://github.com/matplotlib/matplotlib/pull/3322.diff
# unix2dos 3322.diff (if on windows to fix line endings)
patch -p2 < 3322.diff

Qt4 backend

Rudolf Höfler changed the appearance of the subplottool. All sliders are vertically arranged now, buttons for tight layout and reset were added. Furthermore, the the subplottool is now implemented as a modal dialog. It was previously a QMainWindow, leaving the SPT open if one closed the plot window.

In the figure options dialog one can now choose to (re-)generate a simple automatic legend. Any explicitly set legend entries will be lost, but changes to the curves’ label, linestyle, et cetera will now be updated in the legend.

Interactive performance of the Qt4 backend has been dramatically improved under windows.

The mapping of key-signals from Qt to values matplotlib understands was greatly improved (For both Qt4 and Qt5).

Cairo backends

The Cairo backends are now able to use the cairocffi bindings which are more actively maintained than the pycairo bindings.

Gtk3Agg backend

The Gtk3Agg backend now works on Python 3.x, if the cairocffi bindings are installed.

PDF backend

Added context manager for saving to multi-page PDFs.

5.1.8 Text

Text URLs supported by SVG backend

The svg backend will now render Text objects’ url as a link in output SVGs. This allows one to make clickable text in saved figures using the url kwarg of the Text class.

Anchored sizebar font

Added the fontproperties kwarg to AnchoredSizeBar to control the font properties.
5.1.9 Sphinx extensions

The :context: directive in the plot_directive Sphinx extension can now accept an optional reset setting, which will cause the context to be reset. This allows more than one distinct context to be present in documentation. To enable this option, use :context: reset instead of :context: any time you want to reset the context.

5.1.10 Legend and PathEffects documentation

The Legend guide and Path effects guide have both been updated to better reflect the full potential of each of these powerful features.

5.1.11 Widgets

Span Selector

Added an option span_stays to the SpanSelector which makes the selector rectangle stay on the axes after you release the mouse.

5.1.12 GAE integration

Matplotlib will now run on google app engine.

5.2 new in matplotlib-1.3

5.2.1 New in 1.3.1

1.3.1 is a bugfix release, primarily dealing with improved setup and handling of dependencies, and correcting and enhancing the documentation.

The following changes were made in 1.3.1 since 1.3.0.

Enhancements

- Added a context manager for creating multi-page pdfs (see matplotlib.backends.backend_pdf.PdfPages).
- The WebAgg backend should now have lower latency over heterogeneous Internet connections.

Bug fixes

- Histogram plots now contain the endline.
- Fixes to the Molleweide projection.
• Handling recent fonts from Microsoft and Macintosh-style fonts with non-ascii metadata is improved.
• Hatching of fill between plots now works correctly in the PDF backend.
• Tight bounding box support now works in the PGF backend.
• Transparent figures now display correctly in the Qt4Agg backend.
• Drawing lines from one subplot to another now works.
• Unit handling on masked arrays has been improved.

Setup and dependencies

• Now works with any version of pyparsing 1.5.6 or later, without displaying hundreds of warnings.
• Now works with 64-bit versions of Ghostscript on MS-Windows.
• When installing from source into an environment without Numpy, Numpy will first be downloaded and built and then used to build matplotlib.
• Externally installed backends are now always imported using a fully-qualified path to the module.
• Works with newer version of wxPython.
• Can now build with a PyCXX installed globally on the system from source.
• Better detection of Gtk3 dependencies.

Testing

• Tests should now work in non-English locales.
• PEP8 conformance tests now report on locations of issues.

5.2.2 New plotting features

xkcd-style sketch plotting

To give your plots a sense of authority that they may be missing, Michael Droettboom (inspired by the work of many others in PR #1329) has added an xkcd-style sketch plotting mode. To use it, simply call matplotlib.pyplot.xkcd() before creating your plot. For really fine control, it is also possible to modify each artist’s sketch parameters individually with matplotlib.artist.Artist.set_sketch_params().
"Stove Ownership" from xkcd by Randall Monroe
5.2.3 Updated Axes3D.contour methods

Damon McDougall updated the `tricontour()` and `tricontourf()` methods to allow 3D contour plots on arbitrary unstructured user-specified triangulations.
New eventplot plot type

Todd Jennings added a `eventplot()` function to create multiple rows or columns of identical line segments.
As part of this feature, there is a new `EventCollection` class that allows for plotting and manipulating rows or columns of identical line segments.

**Triangular grid interpolation**

Geoffroy Billotey and Ian Thomas added classes to perform interpolation within triangular grids: (`LinearTriInterpolator` and `CubicTriInterpolator`) and a utility class to find the triangles in which points lie (`TrapezoidMapTriFinder`). A helper class to perform mesh refinement and smooth contouring was also added (`UniformTriRefiner`). Finally, a class implementing some basic tools for triangular mesh improvement was added (`TriAnalyzer`).
Baselines for stackplot

Till Stensitzki added non-zero baselines to `stackplot()`. They may be symmetric or weighted.
Rectangular colorbar extensions

Andrew Dawson added a new keyword argument `extendrect` to `colorbar()` to optionally make colorbar extensions rectangular instead of triangular.

More robust boxplots

Paul Hobson provided a fix to the `boxplot()` method that prevent whiskers from being drawn inside the box for oddly distributed data sets.

Calling `subplot()` without arguments

A call to `subplot()` without any arguments now acts the same as `subplot(111)` or `subplot(1,1,1)` – it creates one axes for the whole figure. This was already the behavior for both `axes()` and `subplots()`, and now this consistency is shared with `subplot()`. 
5.2.4 Drawing

Independent alpha values for face and edge colors

Wes Campagne modified how `Patch` objects are drawn such that (for backends supporting transparency) you can set different alpha values for faces and edges, by specifying their colors in RGBA format. Note that if you set the alpha attribute for the patch object (e.g. using `set_alpha()` or the `alpha` keyword argument), that value will override the alpha components set in both the face and edge colors.

Path effects on lines

Thanks to Jae-Joon Lee, path effects now also work on plot lines.

Easier creation of colormap and normalizer for levels with colors

Phil Elson added the `matplotlib.colors.from_levels_and_colors()` function to easily create a colormap and normalizer for representation of discrete colors for plot types such as `matplotlib.pyplot.pcolormesh()`, with a similar interface to that of `contourf()`.

Full control of the background color

Wes Campagne and Phil Elson fixed the Agg backend such that PNGs are now saved with the correct background color when `fig.patch.get_alpha()` is not 1.

Improved `bbox_inches="tight"` functionality

Passing `bbox_inches="tight"` through to `plt.save()` now takes into account all artists on a figure - this was previously not the case and led to several corner cases which did not function as expected.
Initialize a rotated rectangle

Damon McDougall extended the Rectangle constructor to accept an angle kwarg, specifying the rotation of a rectangle in degrees.

5.2.5 Text

Anchored text support

The svg and pgf backends are now able to save text alignment information to their output formats. This allows to edit text elements in saved figures, using Inkscape for example, while preserving their intended position. For svg please note that you’ll have to disable the default text-to-path conversion (mpl.rc(‘svg’, fonttype=’none’)).

Better vertical text alignment and multi-line text

The vertical alignment of text is now consistent across backends. You may see small differences in text placement, particularly with rotated text.

If you are using a custom backend, note that the draw_text renderer method is now passed the location of the baseline, not the location of the bottom of the text bounding box.

Multi-line text will now leave enough room for the height of very tall or very low text, such as superscripts and subscripts.

Left and right side axes titles

Andrew Dawson added the ability to add axes titles flush with the left and right sides of the top of the axes using a new keyword argument loc to title().

Improved manual contour plot label positioning

Brian Mattern modified the manual contour plot label positioning code to interpolate along line segments and find the actual closest point on a contour to the requested position. Previously, the closest path vertex was used, which, in the case of straight contours was sometimes quite distant from the requested location. Much more precise label positioning is now possible.

5.2.6 Configuration (rcParams)

Quickly find rcParams

Phil Elson made it easier to search for rcParameters by passing a valid regular expression to matplotlib.RcParams.find_all(). matplotlib.RcParams now also has a pretty repr and str representation so that search results are printed prettily:
>>> import matplotlib
>>> print(matplotlib.rcParams.find_all('\.size'))
RcParams({'font.size': 12,
        'xtick.major.size': 4,
        'xtick.minor.size': 2,
        'ytick.major.size': 4,
        'ytick.minor.size': 2})

**axes.xmargin and axes.ymargin added to rcParams**

rcParam values (axes.xmargin and axes.ymargin) were added to configure the default margins used. Previously they were hard-coded to default to 0, default value of both rcParam values is 0.

**Changes to font rcParams**

The font.* rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.

**savefig.jpeg_quality added to rcParams**

rcParam value savefig.jpeg_quality was added so that the user can configure the default quality used when a figure is written as a JPEG. The default quality is 95; previously, the default quality was 75. This change minimizes the artifacting inherent in JPEG images, particularly with images that have sharp changes in color as plots often do.

### 5.2.7 Backends

**WebAgg backend**

Michael Droettboom, Phil Elson and others have developed a new backend, WebAgg, to display figures in a web browser. It works with animations as well as being fully interactive.
Future versions of matplotlib will integrate this backend with the IPython notebook for a fully web browser based plotting frontend.

**Remember save directory**

Martin Spacek made the save figure dialog remember the last directory saved to. The default is configurable with the new `savefig.directory` rcParam in `matplotlibrc`.

### 5.2.8 Documentation and examples

**Numpydoc docstrings**

Nelle Varoquaux has started an ongoing project to convert matplotlib’s docstrings to numpydoc format. See [MEP10](#) for more information.
Example reorganization

Tony Yu has begun work reorganizing the examples into more meaningful categories. The new gallery page is the fruit of this ongoing work. See MEP12 for more information.

Examples now use subplots()

For the sake of brevity and clarity, most of the examples now use the newer subplots(), which creates a figure and one (or multiple) axes object(s) in one call. The old way involved a call to figure(), followed by one (or multiple) subplot() calls.

5.2.9 Infrastructure

Housecleaning

A number of features that were deprecated in 1.2 or earlier, or have not been in a working state for a long time have been removed. Highlights include removing the Qt version 3 backends, and the FltkAgg and Emf backends. See Changes in 1.3.x for a complete list.

New setup script

matplotlib 1.3 includes an entirely rewritten setup script. We now ship fewer dependencies with the tarballs and installers themselves. Notably, pytz, dateutil, pyparsing and six are no longer included with matplotlib. You can either install them manually first, or let pip install them as dependencies along with matplotlib. It is now possible to not include certain subcomponents, such as the unit test data, in the install. See setup.cfg.template for more information.

XDG base directory support

On Linux, matplotlib now uses the XDG base directory specification to find the matplotlibrc configuration file. matplotlibrc should now be kept in config/matplotlib, rather than matplotlib. If your configuration is found in the old location, it will still be used, but a warning will be displayed.

Catch opening too many figures using pyplot

Figures created through pyplot.figure are retained until they are explicitly closed. It is therefore common for new users of matplotlib to run out of memory when creating a large series of figures in a loop without closing them.

matplotlib will now display a RuntimeWarning when too many figures have been opened at once. By default, this is displayed for 20 or more figures, but the exact number may be controlled using the figure.max_open_warning rcParam.
5.3 new in matplotlib 1.2.2

5.3.1 Improved collections

The individual items of a collection may now have different alpha values and be rendered correctly. This also fixes a bug where collections were always filled in the PDF backend.

5.3.2 Multiple images on same axes are correctly transparent

When putting multiple images onto the same axes, the background color of the axes will now show through correctly.

5.4 new in matplotlib-1.2

5.4.1 Python 3.x support

Matplotlib 1.2 is the first version to support Python 3.x, specifically Python 3.1 and 3.2. To make this happen in a reasonable way, we also had to drop support for Python versions earlier than 2.6.

This work was done by Michael Droettboom, the Cape Town Python Users’ Group, many others and supported financially in part by the SAGE project.

The following GUI backends work under Python 3.x: Gtk3Cairo, Qt4Agg, TkAgg and MacOSX. The other GUI backends do not yet have adequate bindings for Python 3.x, but continue to work on Python 2.6 and 2.7, particularly the Qt and QtAgg backends (which have been deprecated). The non-GUI backends, such as PDF, PS and SVG, work on both Python 2.x and 3.x.

Features that depend on the Python Imaging Library, such as JPEG handling, do not work, since the version of PIL for Python 3.x is not sufficiently mature.

5.4.2 PGF/TikZ backend

Peter Würtz wrote a backend that allows matplotlib to export figures as drawing commands for LaTeX. These can be processed by PdLaTeX, XeLaTeX or LuaLaTeX using the PGF/TikZ package. Usage examples and documentation are found in *Typesetting With XeLaTeX/LuaLaTeX*. 

Chapter 5. What’s new in matplotlib
5.4.3 Locator interface

Philip Elson exposed the intelligence behind the tick Locator classes with a simple interface. For instance, to get no more than 5 sensible steps which span the values 10 and 19.5:

```python
>>> import matplotlib.ticker as mticker
>>> locator = mticker.MaxNLocator(nbins=5)
>>> print(locator.tick_values(10, 19.5))
```

5.4.4 Tri-Surface Plots

Damon McDougall added a new plotting method for the mplot3d toolkit called `plot_trisurf()`.
5.4.5 Control the lengths of colorbar extensions

Andrew Dawson added a new keyword argument `extendfrac` to `colorbar()` to control the length of minimum and maximum colorbar extensions.
5.4.6 Figures are picklable

Philip Elson added an experimental feature to make figures picklable for quick and easy short-term storage of plots. Pickle files are not designed for long term storage, are unsupported when restoring a pickle saved in another matplotlib version and are insecure when restoring a pickle from an untrusted source. Having said this, they are useful for short term storage for later modification inside matplotlib.

5.4.7 Set default bounding box in matplotlibrc

Two new defaults are available in the matplotlibrc configuration file: savefig.bbox, which can be set to ‘standard’ or ‘tight’, and savefig.pad_inches, which controls the bounding box padding.

5.4.8 New Boxplot Functionality

Users can now incorporate their own methods for computing the median and its confidence intervals into the boxplot() method. For every column of data passed to boxplot, the user can specify an accompanying median and confidence interval.
5.4.9 New RC parameter functionality

Matthew Emmett added a function and a context manager to help manage RC parameters: `rc_file()` and `rc_context`. To load RC parameters from a file:

```python
>>> mpl.rc_file('mpl.rc')
```

To temporarily use RC parameters:

```python
>>> with mpl.rc_context(fname='mpl.rc', rc={'text.usetex': True}):
    ...;
```

5.4.10 Streamplot

Tom Flannaghan and Tony Yu have added a new `streamplot()` function to plot the streamlines of a vector field. This has been a long-requested feature and complements the existing `quiver()` function for plotting vector fields. In addition to simply plotting the streamlines of the vector field, `streamplot()` allows users to map the colors and/or line widths of the streamlines to a separate parameter, such as the speed or local intensity of the vector field.
5.4. new in matplotlib-1.2
5.4.11 New hist functionality

Nic Eggert added a new stacked kwarg to `hist()` that allows creation of stacked histograms using any of the histogram types. Previously, this functionality was only available by using the barstacked histogram type. Now, when `stacked=True` is passed to the function, any of the histogram types can be stacked. The barstacked histogram type retains its previous functionality for backwards compatibility.

5.4.12 Updated shipped dependencies

The following dependencies that ship with matplotlib and are optionally installed alongside it have been updated:

- pytz 2012d
- dateutil 1.5 on Python 2.x, and 2.1 on Python 3.x

5.4.13 Face-centred colors in tripcolor plots

Ian Thomas extended `tripcolor()` to allow one color value to be specified for each triangular face rather than for each point in a triangulation.
tripcolor of Delaunay triangulation, flat shading
pcolor of Delaunay triangulation, gouraud shading
5.4.14 Hatching patterns in filled contour plots, with legends

Phil Elson added support for hatching to `contourf()`, together with the ability to use a legend to identify contoured ranges.
5.4.15 Known issues in the matplotlib-1.2 release

- When using the Qt4Agg backend with IPython 0.11 or later, the save dialog will not display. This should be fixed in a future version of IPython.

5.5 new in matplotlib-1.1

5.5.1 Sankey Diagrams

Kevin Davies has extended Yannick Copin’s original Sankey example into a module (sankey) and provided new examples (api example code: sankey_demoBasics.py, api example code: sankey_demoLinks.py, api example code: sankey_demoRankine.py).
5.5.2 Animation

Ryan May has written a backend-independent framework for creating animated figures. The animation module is intended to replace the backend-specific examples formerly in the Matplotlib Examples listings. Examples using the new framework are in animation Examples; see the entrancing double pendulum which uses matplotlib.animation.Animation.save() to create the movie below.
This should be considered as a beta release of the framework; please try it and provide feedback.

5.5.3 Tight Layout

A frequent issue raised by users of matplotlib is the lack of a layout engine to nicely space out elements of the plots. While matplotlib still adheres to the philosophy of giving users complete control over the placement of plot elements, Jae-Joon Lee created the `tight_layout` module and introduced a new command `tight_layout()` to address the most common layout issues.

The usage of this functionality can be as simple as

```
plt.tight_layout()
```

and it will adjust the spacing between subplots so that the axis labels do not overlap with neighboring subplots. A *Tight Layout guide* has been created to show how to use this new tool.
5.5.4 PyQT4, PySide, and IPython

Gerald Storer made the Qt4 backend compatible with PySide as well as PyQT4. At present, however, PySide does not support the PyOS_InputHook mechanism for handling gui events while waiting for text input, so it cannot be used with the new version 0.11 of IPython. Until this feature appears in PySide, IPython users should use the PyQT4 wrapper for QT4, which remains the matplotlib default.

An rcParam entry, “backend.qt4”, has been added to allow users to select PyQt4, PyQt4v2, or PySide. The latter two use the Version 2 Qt API. In most cases, users can ignore this rcParam variable; it is available to aid in testing, and to provide control for users who are embedding matplotlib in a PyQt4 or PySide app.

5.5.5 Legend

Jae-Joon Lee has improved plot legends. First, legends for complex plots such as stem() plots will now display correctly. Second, the ‘best’ placement of a legend has been improved in the presence of NANS.

See the Legend guide for more detailed explanation and examples.

5.5.6 mplot3d

In continuing the efforts to make 3D plotting in matplotlib just as easy as 2D plotting, Ben Root has made several improvements to the mplot3d module.
- **Axes3D** has been improved to bring the class towards feature-parity with regular Axes objects
- Documentation for *mplot3d* was significantly expanded
- Axis labels and orientation improved
- Most 3D plotting functions now support empty inputs
- Ticker offset display added:

![3D plot](image)

- *contourf()* gains *zdir* and *offset* kwargs. You can now do this:
5.5.7 Numerix support removed

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

5.5.8 Markers

The list of available markers for plot() and scatter() has now been merged. While they were mostly similar, some markers existed for one function, but not the other. This merge did result in a conflict for the ‘d’ diamond marker. Now, ‘d’ will be interpreted to always mean “thin” diamond while ‘D’ will mean “regular” diamond.

Thanks to Michael Droettboom for this effort.

5.5.9 Other improvements

- Unit support for polar axes and arrow()
- PolarAxes gains getters and setters for “theta_direction”, and “theta_offset” to allow for theta to go in either the clock-wise or counter-clockwise direction and to specify where zero degrees should be placed. set_theta_zero_location() is an added convenience function.
• Fixed error in argument handling for tri-functions such as `tripcolor()`
• `axes.labelweight` parameter added to rcParams.
• For `imshow()`, `interpolation='nearest'` will now always perform an interpolation. A “none” option has been added to indicate no interpolation at all.
• An error in the Hammer projection has been fixed.
• `clabel` for `contour()` now accepts a callable. Thanks to Daniel Hyams for the original patch.
• Jae-Joon Lee added the HBox and VBox classes.
• Christoph Gohlke reduced memory usage in `imshow()`.
• `scatter()` now accepts empty inputs.
• The behavior for ‘symlog’ scale has been fixed, but this may result in some minor changes to existing plots. This work was refined by ssyr.
• Peter Butterworth added named figure support to `figure()`.
• Michiel de Hoon has modified the MacOSX backend to make its interactive behavior consistent with the other backends.
• Pim Schellart added a new colormap called “cubehelix”. Sameer Grover also added a colormap called “coolwarm”. See it and all other colormaps here.
• Many bug fixes and documentation improvements.

5.6 new in matplotlib-1.0

5.6.1 HTML5/Canvas backend

Simon Ratcliffe and Ludwig Schwardt have released an HTML5/Canvas backend for matplotlib. The backend is almost feature complete, and they have done a lot of work comparing their html5 rendered images with our core renderer Agg. The backend features client/server interactive navigation of matplotlib figures in an html5 compliant browser.

5.6.2 Sophisticated subplot grid layout

Jae-Joon Lee has written gridspec, a new module for doing complex subplot layouts, featuring row and column spans and more. See Customizing Location of Subplot Using GridSpec for a tutorial overview.
5.6.3 Easy pythonic subplots

Fernando Perez got tired of all the boilerplate code needed to create a figure and multiple subplots when using the matplotlib API, and wrote a `subplots()` helper function. Basic usage allows you to create the figure and an array of subplots with numpy indexing (starts with 0). e.g.:

```python
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3])  # upper, left

See `pylab_examples example code: subplots_demo.py` for several code examples.

5.6.4 Contour fixes and and triplot

Ian Thomas has fixed a long-standing bug that has vexed our most talented developers for years. `contourf()` now handles interior masked regions, and the boundaries of line and filled contours coincide.

Additionally, he has contributed a new module `tri` and helper function `triplot()` for creating and plotting unstructured triangular grids.
triplot of Delaunay triangulation
5.6.5 multiple calls to show supported

A long standing request is to support multiple calls to `show()`. This has been difficult because it is hard to get consistent behavior across operating systems, user interface toolkits and versions. Eric Firing has done a lot of work on rationalizing show across backends, with the desired behavior to make show raise all newly created figures and block execution until they are closed. Repeated calls to show should raise newly created figures since the last call. Eric has done a lot of testing on the user interface toolkits and versions and platforms he has access to, but it is not possible to test them all, so please report problems to the mailing list and bug tracker.

5.6.6 mplot3d graphs can be embedded in arbitrary axes

You can now place an mplot3d graph into an arbitrary axes location, supporting mixing of 2D and 3D graphs in the same figure, and/or multiple 3D graphs in a single figure, using the "projection" keyword argument to `add_axes` or `add_subplot`. Thanks Ben Root.
5.6.7 tick_params

Eric Firing wrote tick_params, a convenience method for changing the appearance of ticks and tick labels. See pyplot function `tick_params()` and associated Axes method `tick_params()`.

5.6.8 Lots of performance and feature enhancements

- Faster magnification of large images, and the ability to zoom in to a single pixel
- Local installs of documentation work better
- Improved “widgets” – mouse grabbing is supported
- More accurate snapping of lines to pixel boundaries
- More consistent handling of color, particularly the alpha channel, throughout the API

5.6.9 Much improved software carpentry

The matplotlib trunk is probably in as good a shape as it has ever been, thanks to improved software carpentry. We now have a buildbot which runs a suite of nose regression tests on every svn commit, auto-generating a set of images and comparing them against a set of known-goods, sending emails to developers on failures
with a pixel-by-pixel image comparison. Releases and release bug fixes happen in branches, allowing active new feature development to happen in the trunk while keeping the release branches stable. Thanks to Andrew Straw, Michael Droettboom and other matplotlib developers for the heavy lifting.

5.6.10 Bugfix marathon

Eric Firing went on a bug fixing and closing marathon, closing over 100 bugs on the bug tracker with help from Jae-Joon Lee, Michael Droettboom, Christoph Gohlke and Michiel de Hoon.

5.7 new in matplotlib-0.99

5.7.1 New documentation


5.7.2 mplot3d

Reinier Heeres has ported John Porter’s mplot3d over to the new matplotlib transformations framework, and it is now available as a toolkit mpl_toolkits.mplot3d (which now comes standard with all mpl installs). See mplot3d Examples and mplot3d tutorial
5.7.3 axes grid toolkit

Jae-Joon Lee has added a new toolkit to ease displaying multiple images in matplotlib, as well as some support for curvilinear grids to support the world coordinate system. The toolkit is included standard with all new mpl installs. See axes_grid Examples and The Matplotlib AxesGrid Toolkit User's Guide.
5.7.4 Axis spine placement

Andrew Straw has added the ability to place “axis spines” – the lines that denote the data limits – in various arbitrary locations. No longer are your axis lines constrained to be a simple rectangle around the figure – you can turn on or off left, bottom, right and top, as well as “detach” the spine to offset it away from the data. See `pylab_examples example code: spine_placement_demo.py` and `matplotlib.spines.Spine`. 
5.8 new in 0.98.4

It’s been four months since the last matplotlib release, and there are a lot of new features and bug-fixes.

Thanks to Charlie Moad for testing and preparing the source release, including binaries for OS X and Windows for python 2.4 and 2.5 (2.6 and 3.0 will not be available until numpy is available on those releases). Thanks to the many developers who contributed to this release, with contributions from Jae-Joon Lee, Michael Droettboom, Ryan May, Eric Firing, Manuel Metz, Jouni K. Seppänen, Jeff Whitaker, Darren Dale, David Kaplan, Michiel de Hoon and many others who submitted patches

5.8.1 Legend enhancements

Jae-Joon has rewritten the legend class, and added support for multiple columns and rows, as well as fancy box drawing. See legend() and matplotlib.legend.Legend.
5.8.2 Fancy annotations and arrows

Jae-Joon has added lots of support to annotations for drawing fancy boxes and connectors in annotations. See `annotate()` and `BoxStyle`, `ArrowStyle`, and `ConnectionStyle`. 
5.8.3 Native OS X backend

Michiel de Hoon has provided a native Mac OSX backend that is almost completely implemented in C. The backend can therefore use Quartz directly and, depending on the application, can be orders of magnitude faster than the existing backends. In addition, no third-party libraries are needed other than Python and NumPy. The backend is interactive from the usual terminal application on Mac using regular Python. It hasn’t been tested with ipython yet, but in principle it should to work there as well. Set `backend : macosx` in your matplotlibrc file, or run your script with:

```
> python myfile.py -dmacosx
```

5.8.4 psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of `psd()` and friends. See `pylab_examples` example code: `psd_demo2.py`, and `pylab_examples` example code: `psd_demo3.py`. The changes should
increase MATLAB compatibility and increase scaling options.

### 5.8.5 Fill between

Added a `fill_between()` function to make it easier to do shaded region plots in the presence of masked data. You can pass an x array and a `ylower` and `yupper` array to fill between, and an optional `where` argument which is a logical mask where you want to do the filling.

```
fill between where
```

### 5.8.6 Lots more

Here are the 0.98.4 notes from the CHANGELOG:

Added mdehoon's native macosx backend from sf patch 2179017 - JDH

Removed the prints in the set_*style commands. Return the list of pretty-printed strings instead - JDH

Some of the changes Michael made to improve the output of the property tables in the rest docs broke of made difficult to use some of the interactive doc helpers, e.g., setp and getp. Having all the rest markup in the ipython shell also confused the docstrings. I added a new rc param docstring.harcopy, to format the docstrings differently for hardcopy and other use. The ArtistInspector
could use a little refactoring now since there is duplication of effort between the rest out put and the non-rest output - JDH

Updated spectral methods (psd, csd, etc.) to scale one-sided densities by a factor of 2 and, optionally, scale all densities by the sampling frequency. This gives better MATLAB compatibility. -RM

Fixed alignment of ticks in colorbars. -MGD

drop the deprecated "new" keyword of np.histogram() for numpy 1.2 or later. -JJL

Fixed a bug in svg backend that new_figure_manager() ignores keywords arguments such as figsize, etc. -JJL

Fixed a bug that the handlelength of the new legend class set too short when numpoints=1 -JJL

Added support for data with units (e.g., dates) to Axes.fill_between. -RM

Added fancybox keyword to legend. Also applied some changes for better look, including baseline adjustment of the multiline texts so that it is center aligned. -JJL

The transmuter classes in the patches.py are reorganized as subclasses of the Style classes. A few more box and arrow styles are added. -JJL

Fixed a bug in the new legend class that didn't allowed a tuple of coordinate values as loc. -JJL

Improve checks for external dependencies, using subprocess (instead of deprecated popen*) and distutils (for version checking) - DSD

Reimplementation of the legend which supports baseline alignment, multi-column, and expand mode. - JJL

Fixed histogram autoscaling bug when bins or range are given explicitly (fixes Debian bug 503148) - MM

Added rcParam axes.unicode_minus which allows plain hyphen for minus when False - JDH

Added scatterpoints support in Legend. patch by Erik Tollerud - JJL

Fix crash in log ticking. - MGD

Added static helper method BrokenHBarCollection.span_where and Axes/pyplot method fill_between. See

5.8. new in 0.98.4
Add `x_isdata` and `y_isdata` attributes to Artist instances, and use them to determine whether either or both coordinates are used when updating `datalim`. This is used to fix autoscaling problems that had been triggered by `axhline`, `axhspan`, `axvline`, `axvspan`. - EF

Update the `psd()`, `csd()`, `cohere()`, and `specgram()` methods of Axes and the `csd()` `cohere()`, and `specgram()` functions in `mlab` to be in sync with the changes to `psd()`. In fact, under the hood, these all call the same core to do computations. - RM

Add 'pad_to' and 'sides' parameters to `mlab.psd()` to allow controlling of zero padding and returning of negative frequency components, respectively. These are added in a way that does not change the API. - RM

Fix handling of `c` kwarg by `scatter`; generalize `is_string_like` to accept `numpy` and `numpy.ma` string array scalars. - RM and EF

Fix a possible EINTR problem in `dviread`, which might help when saving pdf files from the qt backend. - JKS

Fix bug with zoom to rectangle and twin axes - MGD

Added Jae Joon's fancy arrow, box and annotation enhancements -- see examples/pylab_examples/annotation_demo2.py

Autoscaling is now supported with shared axes - EF

Fixed exception in `dviread` that happened with Minion - JKS

`set_xlim`, `ylim` now return a copy of the viewlim array to avoid modify inplace surprises

Added image thumbnail generating function `matplotlib.image.thumbnail`. See examples/misc/image_thumbnail.py - JDH

Applied scatleg patch based on ideas and work by Erik Tollerud and Jae-Joon Lee. - MM

Fixed bug in pdf backend: if you pass a file object for output instead of a filename, e.g., in a wep app, we now flush the object at the end. - JKS

Add path simplification support to paths with gaps. - EF

Fix problem with AFM files that don't specify the font's full name or family name. - JKS

Added 'scilimits' kwarg to Axes.ticklabel_format() method, for easy access to the set_powerlimits method of the major
ScalarFormatter. - EF

Experimental new kwarg borderpad to replace pad in legend, based on suggestion by Jae-Joon Lee. - EF

Allow spy to ignore zero values in sparse arrays, based on patch by Tony Yu. Also fixed plot to handle empty data arrays, and fixed handling of markers in figlegend. - EF

Introduce drawstyles for lines. Transparently split linestyles like 'steps--' into drawstyle 'steps' and linestyle '--'. Legends always use drawstyle 'default'. - MM

Fixed quiver and quiverkey bugs (failure to scale properly when resizing) and added additional methods for determining the arrow angles - EF

Fix polar interpolation to handle negative values of theta - MGD

Reorganized cbook and mlab methods related to numerical calculations that have little to do with the goals of those two modules into a separate module numerical_methods.py Also, added ability to select points and stop point selection with keyboard in ginput and manual contour labeling code. Finally, fixed contour labeling bug. - DMK

Fix backtick in Postscript output. - MGD

[ 2089958 ] Path simplification for vector output backends
Leverage the simplification code exposed through path_to_polygons to simplify certain well-behaved paths in the vector backends (PDF, PS and SVG). "path.simplify" must be set to True in matplotlibrc for this to work. - MGD

Add "filled" kwarg to Path.intersects_path and Path.intersects_bbox. - MGD

Changed full arrows slightly to avoid an xpdf rendering problem reported by Friedrich Hagedorn. - JKS

Fix conversion of quadratic to cubic Bezier curves in PDF and PS backends. Patch by Jae-Joon Lee. - JKS

Added 5-point star marker to plot command q- EF

Fix hatching in PS backend - MGD

Fix log with base 2 - MGD

Added support for bilinear interpolation in NonUniformImage; patch by Gregory Lielens. - EF

Added support for multiple histograms with data of 5.8. new in 0.98.4
different length - MM

Fix step plots with log scale - MGD

Fix masked arrays with markers in non-Agg backends - MGD

Fix clip_on kwarg so it actually works correctly - MGD

Fix locale problems in SVG backend - MGD

fix quiver so masked values are not plotted - JSW

improve interactive pan/zoom in qt4 backend on windows - DSD

Fix more bugs in NaN/inf handling. In particular, path
simplification (which does not handle NaNs or infs) will be turned
off automatically when infs or NaNs are present. Also masked
arrays are now converted to arrays with NaNs for consistent
handling of masks and NaNs - MGD and EF

Added support for arbitrary rasterization resolutions to the SVG
backend. - MW
CHAPTER
SIX

GITHUB STATS

GitHub stats for 2013/07/31 - 2014/10/18 (tag: v1.3.0)
These lists are automatically generated, and may be incomplete or contain duplicates.
The following 169 authors contributed 2105 commits.

- Adam Heck
- Adrian Price-Whelan
- Alex Loew
- Alistair Muldal
- Andrea Bedini
- Andreas Wallner
- Andrew Dawson
- Andrew Merrill
- Aseem Bansal
- Behram Mistree
- Ben Cohen
- Ben Gamari
- Ben Keller
- Ben Root
- Benjamin Reedlunn
- Brandon Liu
- CJ Carey
- Cameron Davidson-Pilon
- Carissa Brittain
- Carwyn Pelley
- Chris Beaumont
• Chris G
• Christoph Gohlke
• Christoph Hoffmann
• Cimarron Mittelsteadt
• Damon McDougall
• Daniel O’Connor
• Dara Adib
• David Anderson
• Dean Malmgren
• Dmitry Lupyan
• Donald Seo
• Elliott Sales de Andrade
• Eric Firing
• Eugene Yurtsev
• Federico Ariza
• Felipe
• Filipe
• Francesco Montesano
• Francis Colas
• Geoffroy Billotey
• Gregory Ashton
• Guillaume Gay
• Gustavo Braganca
• Hans Meine
• Hans Moritz Günther
• Ian Thomas
• Jae-Joon Lee
• Jake Vanderplas
• James Makela
• Jan Schulz
• Jason Grout
• Jason Miller
• Jens Hedegaard Nielsen
• Joe Kington
• Joel B. Mohler
• Jorrit Wronski
• José Ricardo
• Jouni K. Seppänen
• Julian Taylor
• JulianCienfuegos
• Katy Huff
• Kevin Chan
• Kevin Keating
• Kristen M. Thyng
• Larry Bradley
• Lennart Fricke
• Leo Singer
• Loïc Séguin-C
• Magnus Nord
• Maksym P
• Manuel GOACOLOU
• Marianne Corvellec
• Markus Roth
• Martin Dengler
• Martin Fitzpatrick
• Martin Spacek
• Matt Klein
• Matt Terry
• Matthew Brett
• Matthias Bussonnier
• Matthieu Caneill
• Matěj Týč
• Michael
• Michael Droettboom
• Michiel de Hoon
• Michka Popoff
• Mikhail Korobov
• MinRK
• Nelle Varoquaux
• Nic Eggert
• Nicolas P. Rougier
• Oliver Willekens
• Patrick Marsh
• Paul
• Paul Hobson
• Paul Ivanov
• Per Parker
• Peter Iannucci
• Peter Würtz
• Phil Elson
• Pierre Haessig
• Puneeth Chaganti
• Remi Rampin
• Richard Hattersley
• Ricky
• Robert Johansson
• Rohan Walker
• Roland Wirth
• RutgerK
• Ryan Blomberg
• Ryan D’Souza
• Ryan May
• Scott Lasley
• Scott Lawrence
• Scott Stevenson
• Sergey Kholodilov
• Silviu Tantos
• Simon Gibbons
• Thomas A Caswell
• Thomas Hisch
• Thomas Robitaille
• Till Stensitzki
• Timo Vanwynsberghe
• Tobias Megies
• Todd Jennings
• Tony S Yu
• Tor Colvin
• Trevor Bekolay
• Vadim Markovtsev
• Valentin Haenel
• Viktor Kerkez
• Vlad Seghete
• Wieland Hoffmann
• William Manley
• Yaron de Leeuw
• anykraus
• arokem
• aszilagyi
• blackw1ng
• blah blah
• captainwhippet
• chebee7i
• danielballan
• davidovitch
• daydreamt
• donald
• endolith
• fardal
We closed a total of 1964 issues, 622 pull requests and 1342 regular issues; this is the full list (generated with the script tools/github_stats.py):

Pull Requests (622):

- **PR #3672**: Python3 pep8 fixes
- **PR #3558**: Adds multiple histograms side-by-side example
- **PR #3665**: Remove usage of raw strides member in _backend_gdk.c
- **PR #3309**: Explicitly close read and write of Popen process (latex)
- **PR #3662**: Make all classes new-style.
- **PR #3646**: Remove PyCXX dependency for core extension modules
- **PR #3664**: [examples] pep8 fix e251 e27*
- **PR #3294**: fix typo in figlegend_demo.py
- **PR #3666**: remove print from test
- PR #3638: MNT: slight refactoring of Gcf
- PR #3387: include PySide in qt4agg backend check
- PR #3597: BUG/TST: skip example pep8 if don’t know source path
- PR #3661: Numpy 1.6 fixes
- PR #3635: fix pep8 error classes e20[12] and e22[12] in examples
- PR #3547: Don’t use deprecated numpy APIs
- PR #3640: figure.max_num_figures was renamed to figure.max_open_warning.
- PR #3650: Typo fixes. [backport to doc branch]
- PR #3564: Rcrem validation fix
- PR #3642: TST: know-fail shading tests
- PR #3632: Fix for #3623
- PR #3619: PatchCollection: pass other kwargs for match_original=True
- PR #3629: examples: fix pep8 error class E211
- PR #3622: setup.py creates a zombie C extension called “freetype2”
- PR #3627: Fixed Image and Renderer pickling
- PR #3515: examples: fix pep8 error classes E111 and E113
- PR #3625: animate_decay.py example code is less complicated
- PR #3621: matplotlib.__version__ is now unicode as of 1.4.0
- PR #3620: Revert interactive determination
- PR #3613: Fix problem with legend if data has NaN’s [backport to 1.4.x]
- PR #3313: BUG: 3 fixes for widgets (MultiCrusor, SpanSelector, Slider)
- PR #3496: BUG: fix str vs bytes issue in py3 in ps backend
- PR #3609: Nbagg icons
- PR #3611: Fix spelling error
- PR #3600: BUG: now only set ‘marker’ and ‘color’ attribute of fliers in boxplots
- PR #3560: Updated what's new for nbagg and legend/patheffects docs.
- PR #3594: Unicode decode error [backport to 1.4.x]
- PR #3574: rotation parameter has no effect on RegularPolyCollection
- PR #3409: Win fixes
- PR #3595: Some small doc fixes only relevant on the master branch
- PR #3291: Lightsource enhancements
- PR #3592: Fix crash in picking for zero-length path collection
- PR #3585: merge V1.4.0-doc into v1.4.x
- PR #3566: BUG: don’t assume label in boxplot_stat
- PR #3567: Fixed the differencing of images for the webagg/nbagg backends.
- PR #3571: BUG: deal with empty list passed to boxplot
- PR #3533: BUG: fix handling of flierprop by boxplot
- PR #3514: Ticks on top axis disappear if tick size is too large (when using bbox_inches=’tight’)
- PR #3578: Fixes test to assert instead of print
- PR #3575: Supports locale-specified encoding for rcfile.
- PR #3479: Build dep updates
- PR #3552: Nbagg enhancements
- PR #3559: Install texlive and other dependencies when building docs.
- PR #3555: Typo in comment documentation for example timers.py
- PR #3556: copy/paste corrections in test_backend_qt5
- PR #3545: Provide an informative error message if something goes wrong in setfont [backport to 1.4.x]
- PR #3548: Silence some Sphinx warnings by rewriting docstrings.
- PR #3539: DEP: update six minimum version
- PR #3543: DOC: fix main-page tags
- PR #3524: Bug in AutoDateLocator when dates are in reverse order
- PR #3464: BUG: nbagg py3k compatibility
- PR #3534: BUG: fixes whis over-writing in boxplot_stats
- PR #3535: BUG/DOC: Correct default value listed in docstring
- PR #3369: Added legend.framealpha to rcParams, as mentioned in axes.legend docstring
- PR #3510: Fix setupext [backport to 1.4.x]
- PR #3530: Only insert links to pdfs if we are actually generating these.
- PR #3487: Can not import matplotlib when launching from non-ascii path
- PR #3526: BUG: fix eps corruption when using clipping
- PR #3492: Allow python 3 version of PyCXX
- PR #3521: More doc fixes
- PR #3513: examples: fully automated fixing of E30 pep8 errors
• PR #3461: This fixes a bunch of Sphinx warnings
• PR #3507: general pep8 fixes
• PR #3506: Named colors example, figure size correction [backport to 1.4.0-doc]
• PR #3503: Win fix simple
• PR #3495: BUG : don’t use super(self.__class__, self)
• PR #3501: Bugfix for text.xytext property
• PR #3376: Move widget.{get,set}_active to AxisWidget.
• PR #3419: Better repr for Bboxes.
• PR #3474: call set cursor on zoom/pan toggle [backport to 1.4.x]
• PR #3425: Pep8ify examples
• PR #3477: Better check for required dependency libpng
• PR #3478: BUG : restore back-compatibility of registter_backend
• PR #2900: Remove no-longer-necessary KnownFail for python 3.2.
• PR #3467: Bugfix in mlab for strided views of np.arrays [backport to 1.4.x]
• PR #3469: Fix handling of getSaveFileName to be consistent [backport to 1.4.x]
• PR #3384: Test marker styles
• PR #3456: DOC : add known- bug + fix for QT5 toolbar issue
• PR #3458: DOC : minor import tweaks
• PR #3457: Add Qt5Agg to backends in matplotlibrc.template.
• PR #3422: Use mailmap in github stats
• PR #3429: DOC : added pytz to optional dependencies docs
• PR #3453: Bbox rebase 14
• PR #3427: DOC : added DOI link to citing.html
• PR #3438: Get rid of unused pre python 2.6 code in doc make.py
• PR #3414: Fixes TypeError when installing without freetype
• PR #3428: DOC : add caveat about freetype 2.3
• PR #3430: DOC : fixed markup / boxplot text in whats_new
• PR #3426: DOC : remove piwik code
• PR #3432: Update whats_new.rst
• PR #3415: DOC: fix markup and download link in the Windows install section
• PR #3408: test_text Modify filter warning regex
• PR #3282: Catch warning thrown in Mollweide projection.
• PR #2635: Crash on saving figure if text.usetex is True
• PR #3241: Cast to integer to get rid of numpy warning
• PR #3335: Fix and new test for #3327
• PR #3278: Use libpng-config if available
• PR #3244: Filter warnings in rcparams test (and others)
• PR #3378: BUG: Fixes custom path marker sizing for issue #1980
• PR #3401: enlarge on windows build instructions slightly
• PR #3399: Cherrypick doc changes to v1.4.0
• PR #3397: Install guide tweaks
• PR #3394: DOC : add note about np.matrix and pandas objects
• PR #3390: Move stylelib directory to mpl-data
• PR #3381: BUG : make qApp global before using it
• PR #3380: Remove residual mention of TODO file, which no longer exists.
• PR #3349: DOC : added folders for api_changes and whats_new
• PR #3360: BUG : modified logic on starting qApp
• PR #3372: DOC: Fixed the wording of the deprecation warning
• PR #3363: Identification of freetype when ‘freetype-config –ftversion’ fails.
• PR #3359: PEP8 conformity; removed outcommented code
• PR #3357: backend_qt5.py Don’t use six.u
• PR #3287: DOC: comprehensive rewrite for OSX binary install
• PR #3337: BUG : don’t assign color to flier props if None
• PR #3342: AGG link changed. Also fix other doc warnings
• PR #3262: 1.4.0 RC1: –ftversion vs –version freetype version
• PR #3322: Fixed error with QSizePolicy
• PR #3339: Fix mathmpl images not showing in HTML Help (CHM)
• PR #3331: Restore compatibility with Python 3.2
• PR #3324: Fix #3304.
• PR #3325: ENH: add HTML Help builder option
• PR #3329: Workaround for Sphinx not escaping _ (underscore) for tex output
• PR #3323: Replaced unicode() function by six.text_type
• PR #3301: Colormap choice guidelines in documentation - based on talk at SciPy 2014
• PR #3320: DOC: Fix Malformed table. Text in column margin
• PR #3317: TST: Fix test_animation RuntimeErrors on Windows
• PR #3310: Fix MatplotlibDeprecationWarning: The “loc” positional argument to legend is deprecated
• PR #3312: BUG: fix test error when ghostscript not installed
• PR #3306: BUG: restore allow_rasterization on PolyCollection
• PR #3302: Check for GhostScript in this test which needs it
• PR #3194: Annotate bbox darrow
• PR #3277: MNT: better error handling on determining gs version
• PR #3300: Quiver3d fixes
• PR #3284: BUG: fix _reshape_2D bug with [(n, 1), ..] input
• PR #3296: V1.4.x
• PR #3295: Use the interpolation parameter in make_thumbnail
• PR #3235: Silence some more warnings
• PR #3274: BUG: fixes FontProperties memory leak
• PR #3275: TST: Fix ImportError: No module named ‘mpl_toolkits’
• PR #3250: Fix WindowsError: [Error 32] The process cannot access the file
• PR #3247: Usage faq
• PR #3270: DOC: windows install docs from cgohlke
• PR #3257: MRG: refactor and bugfixes for plot_directive
• PR #3238: OSX install
• PR #3258: Fix various memory leaks discovered through valgrind
• PR #3253: Stop make.py removing generated documentation figs
• PR #3269: Upload artifacts only on main repository.
• PR #3251: Add animation.convert_path setting to matplotlibrc.template
• PR #3266: remove obsolete TODO files
• PR #3261: Get rid of warning about GTK3Agg with python3
• PR #3249: TST: Fix test_backend_ps failures on Windows
• PR #3217: Added some function arguments to the documentation for FuncAnimation
• PR #3243: Fixed backend workflow.
• PR #3246: Fix some hyperlinks in the documentation
• PR #3004: FAQ and unit/ still refers to nxutils
• PR #3239: Fix auto-closing in PolyCollection
• PR #3193: Fix plot directive when used with multiple options.
• PR #3236: Test PEP8 stuff in separate Travis build.
• PR #3188: Np error patch
• PR #3154: whitelist mpl_toolkits tests
• PR #3230: DOC : added note about useoffset rcparam
• PR #3228: DOC : top_level doc-string clean up
• PR #3190: Adding two new styles to mplstyles
• PR #3215: Close files in animation to silence some warning in the test suite on python3
• PR #3237: Fix Collection3D. Fixes legend for scatter3d
• PR #3233: Update numpy version in setup.py
• PR #3227: Whats new cleaning
• PR #3224: Fix lots of warnings in docs/Examples that crash
• PR #3229: DEP : bump min numpy to 1.6
• PR #3222: add reduce to the list of imports from six.moves
• PR #3126: insertion of Annotation class docs into annotate docstring broken
• PR #3221: Fixes #3219 by ignoring pep8 noncomplicant auto-generated file.
• PR #2227: Refactor of top-level doc/README.rst
• PR #3211: Mplot3d/depthshade
• PR #3184: DOC : added warning to doc of get_window_extent
• PR #3165: Bug restore boxplot defaults
• PR #3207: Fix memory leak in tostring_rgba_minimize(). (#3197)
• PR #3210: Fix PEP8 error.
• PR #3203: Make type1font.py work better on Python 3.x
• PR #3155: BUG : fix fetch of freetype version during build
• PR #3192: TST : drop 3.2, add 3.4
• PR #3121: Added ‘PyQt4v2’ to valid values for backend.qt4
• PR #3167: BUG : raise exception in subplot if num out of range
• PR #3208: Add missing import of unichr from six.
• PR #3156: DOC : added whats_new entry for Qt5 backend
• PR #2843: BUGFIX: This change fixes #2475, where contour labels added manually
• PR #3201: Revert “[examples/api] autopep8 + use np.radians/np.degree where appropr...
• PR #3200: Revert “pep8ify more examples in examples/ + use np.radians/np.degrees”
• PR #3174: MNT : replace and deprecated qt4Compat
- PR #3112: BUG: patches.Wedge.set_radius set wrong attribute
- PR #2952: BUG: turned clipping off on pie chart components
- PR #2951: BUG/API: tweaked how AnchoredSizeBar handles font properties
- PR #3157: BLD: fix build on windows
- PR #3189: BUG: use unittest.mock for Python 3.3+
- PR #3045: Use less aggressive garbage collection
- PR #3185: DOC: added details about r/cstride in plot3d
- PR #3182: pep8ify more examples in examples/ + use np.radians/np.degrees
- PR #3181: [examples/api] autopep8 + use np.radians/np.degree where appropriate
- PR #3163: DOC: documented bottom kwarg of hist
- PR #3180: DOC: Fix order of parameters in ax.text docstring.
- PR #3168: DOC: add prominent doc about set_useOffset
- PR #3162: BLD: made tornado an optional external package
- PR #3169: Update pyplotutorial.rst
- PR #3084: Improving plt.hist documentation
- PR #3160: Glade tutorial branch fixed
- PR #3008: Nbagg backend
- PR #3164: fix bad pathing in whats_new.rst
- PR #3159: BUG: fix qt4 backends
- PR #3158: backend_.pgf: Error message for missing latex executable (fix #3051)
- PR #3125: DOC: added annotation example to arrow docstring
- PR #3149: 3dquiver rebranch
- PR #3141: BUG: Fix ‘TypeError: expected bytes, str found’ on Python 3
- PR #3072: Implement backend for PyQt5 + modify Qt4 backends to use Qt5 module via shim
- PR #3153: Avoid floating point sensitivity in trisurf3d test
- PR #3147: Fix doc for sharey keyword in pyplot.subplots.
- PR #3133: Doc cleanup
- PR #3110: BUG: Add Figure.delcolorbar() to fully delete a colorbar
- PR #3131: DOC: sixify unichr
- PR #3132: DOC: added note about maintain ref to widgets
- PR #2927: BUG: don’t use mutable objects as dictionary keys
- PR #3122: DOC: mention Anaconda; clean some old junk out of the FAQ
• PR #3130: Scatter set sizes what’s new
• PR #3127: DOC : added inherited-members to Axes autodoc
• PR #3128: Axes aspect doc
• PR #3103: errorbar: fmt kwarg defaults to None; use ‘none’ to suppress plot call
• PR #3123: DOC : add documentation to Polygon methods
• PR #3120: typo fix
• PR #3099: New animation example (Joy Division’s Unchained Love cover)
• PR #3111: bug fix: check the type of the ‘key’ of the two array ‘r1’ and ‘r2’
• PR #3108: DOC : clarified doc of add_artist
• PR #3107: Bug-fix for issue 3106
• PR #3092: Adds check that rgb sequence is of length 3
• PR #3100: Use autolim kwarg in add_collection to prevent duplication of effort.
• PR #3104: BUG: in Spine.set_position(), preserve most Axis info.
• PR #3101: Streamplot: clean up handling of masks, eliminate warning in test.
• PR #3102: Image: handle images with zero columns or rows.
• PR #2929: clip_on documentation note/warning
• PR #3067: Fix for bug #3029.
• PR #3078: fix argument checks in axis/base.margins
• PR #3089: Fix log hist y-axis minimum with weighted data
• PR #3087: small error in comment
• PR #2996: Violin Plots
• PR #3035: symlog-scale: Remove assert linscale >= 1.
• PR #3077: Invalidate font manager when rcParam family lists change.
• PR #3081: Points to pixels
• PR #3080: Minor fix to commit 24bc071
• PR #3076: Bug: backend_pdf: UnicodeDecodeError: ‘ascii’ codec can’t decode byte 0xe2
• PR #3074: TST : force re-building of font-cache
• PR #2874: Fix for issue #2541 (revised)
• PR #2662: allow slice and fancy indexing to only show some markers
• PR #2855: ENH Added the origin option to spy
• PR #3022: Updating PyQt version checks for v4.10+
• PR #3015: Date stem simplefix
• PR #3017: Do not provide (wrong) mtext instances for pre-layed text blocks (fixes #3000)
• PR #3009: BUG: Showing a BboxImage can cause a segmentation fault
• PR #3061: Add Axes.add_image() for consistency.
• PR #3063: Change EPD links to Enthought Canopy
• PR #3050: Animation example: rain drops
• PR #2898: Fix animation errors
• PR #3031: avoid np.nan values in colors array returned by axes3d._shade_colors
• PR #3038: BUG: expand x/y range in hexbin if singular
• PR #3018: Fix documentation of entropy function
• PR #3036: Unicode fixes
• PR #2871: Add a colorblind friendly heatmap.
• PR #2879: BLD: adjust min six version to 1.3
• PR #3037: DEP: removed levypdf from mlab
• PR #3025: mpl issue: #2974 - documentation corrected
• PR #3030: Fix minor typo in customisation docs
• PR #2947: Re-Generate legend, through apply_callback/Apply
• PR #3014: BUG: improved input clean up in Axes.{h|v}line
• PR #2771: Fix font family lookup calculation
• PR #3007: #3005 - Removed ‘ipython -pylab’ references
• PR #2946: remove .rect member (clashes with QWidget)
• PR #2837: EXP: turn of clipping in spine example
• PR #2772: BUG: instantiate fall-back writer
• PR #2922: ENH: add flag to box_plot and bxp to manage (or not) xticks
• PR #2950: DOC: edits to optional dependencies
• PR #2995: Added ‘interpolation_none_vs_nearest’ example, without .DS_store files
• PR #3002: BUG/DOC: fix bad merge of INSTALL
• PR #2993: Avoid a null-pointer dereference in _tri.cpp
• PR #2994: Minor fixes in _macosx.m
• PR #2997: Disable copying of C++ classes with nontrivial destructors
• PR #2992: Remove a few dead assignments
• PR #2991: Silence some compiler warnings related to ft2font
• PR #2989: Don’t call Py_DECREF on null in _ttconv.cpp
Matplotlib, Release 1.4.3

- PR #2984: small error in install faq
- PR #2829: (fix #2097) PGF: get fonts from fc-list, use builtin fonts for tests
- PR #2913: Allow :context: directive to take ‘reset’ option. Fixes #2892.
- PR #2914: Don’t close figure if context and apply_reparams are both set.
- PR #2983: DOC/BUG : fixed sphinx markup
- PR #2981: TST: __spec__ (an import-related variable for modules) was added in pyth...
- PR #2978: BUG: EllipseCollection: fix transform error
- PR #2968: BUG: Fix the triangular marker rendering error.
- PR #2966: axvline doc typo fix
- PR #2962: py3k fix
- PR #2739: DOC : improved extent of imshow doc
- PR #2960: PEP8 : making pep8 happy again
- PR #2836: DOC/ENH : get/set_size_inches
- PR #2948: DOC : added missing doc changes from #2844
- PR #1204: Add power-law normalization
- PR #2452: Fixed issues with errorbar limits
- PR #2955: PEP8 : add missing line to un-break build
- PR #2926: BUG: Removes iteration over locals (no-no) in mathtext
- PR #2915: Consistency of the radius argument for Path.points_in_path
- PR #2939: Fixes a bug in drawing bitmap images in the macosx backend for handling device scaling
- PR #2949: CLN : removed version check that required numpy > 1.2
- PR #2848: DOC : removed line about un-needed dependencies in Windows
- PR #2940: Fix some documentation mistakes
- PR #2933: #2897 Adding tests for pie ccw. Issue 2897
- PR #2923: Issue 2899
- PR #2930: Cranky pep8
- PR #2847: DOC : add link to plt.subplots from Figure.add_subplot
- PR #2906: Fix Cairo text on Python3 with pycairo
- PR #2920: fix six check message
- PR #2912: Fix paths in doc which are searched for matplotlibrc (XDG).
- PR #2735: Fixes issue #966: When appending the new axes, there is a bug where it
- PR #2911: text_axes missing cleanups

Chapter 6. Github stats
• PR #2834: WebAgg: Fix IPython detection. Fix encoding error on Python 3
• PR #2853: counterclock parameter for pie
• PR #1664: Support for skewed transforms
• PR #2844: BUG : Qt repaint workaround on windows
• PR #2895: typos: s/cooordinate/coordinate & s/contols/controls
• PR #2875: Fix for issue #2872. Skip NaN’s in draw_path_collection.
• PR #2887: fix a bug introduced in c998561d6cc1236
• PR #2884: Fixed the failing tests on master.
• PR #2851: Fix positional/kwarg handling of the Z argument
• PR #2852: AttributeError: ‘module’ object has no attribute ‘next’
• PR #2860: Fix subprocess.CalledProcessError on Google App Engine
• PR #2865: WebAgg: raise WebAggApplication.started flag before blocking
• PR #2867: GTK3 backend: implemented FigureCanvasBase.resize_event()
• PR #2858: BUG: colorbar autoscaling now ensures a finite range of values
• PR #2849: WebAgg issue - Uncaught SyntaxError: Unexpected token &
• PR #2854: DOC hist is not cumulative by default
• PR #2825: WebAgg: extracted figure_div style into css and changed layout
• PR #2444: Fixed bad vector transforms.
• PR #2731: 2d padding
• PR #2846: Fix bug in horizontal step histograms (#2830)
• PR #2819: DOC: clarified docstring for cbook.boxplot_stats
• PR #2835: quiver: handle autoscaling with quiverkey when animated
• PR #2838: TST : make 3.2 pass again
• PR #2826: GTK3 backend: Replaced deprecated GOobject calls with GLib
• PR #2805: ENH: Updated inset locator axes to return a HostAxes by default
• PR #2807: Python 3 METH_VARARGS with METH_KEYWORDS
• PR #2821: DOC: point downloads at the matplotlib downloads
• PR #2813: GTK3Agg backend: Only convert the cairo context to a cairocffi context o...
• PR #2801: Named colors example
• PR #2784: Scipy2013 Sprint: Cleaning F/C example
• PR #2798: Added remove methods for legends in figure and axes objects
• PR #2799: Xdg message repr
• PR #2781: Triplot returns the artist it adds.
• PR #2774: changed the text of INSTALL to be correct about what external
• PR #2788: MEP12: Clean-up line and marker demos
• PR #2787: Empty event loop
• PR #2779: remove old animation examples.
• PR #2794: fix typo in documentation
• PR #2793: missing mask for scroll event
• PR #2780: ENH : improve error invalid error message for subplot
• PR #2782: BUG: quiverkey must set the vector figure attribute
• PR #2389: table.py: fix issue when specifying both column header text and color
• PR #2755: Fixes legend.get_children() to actually return the real children of
• PR #2599: Create interpolation_methods.py
• PR #2621: Simplify and fix dpi handling in tight_bbox
• PR #2752: Make standardization of input optional in mlab.PCA
• PR #2726: Don’t snap pcolor
• PR #2732: AttributeError: ‘Patch3DCollection’ object has no attribute ‘set_sizes’
• PR #2442: Rewrite of the entire legend documentation, including tidy ups of code and style to all things “legend”.
• PR #2746: ENH : added warning on annotate
• PR #2675: clip_on = False does not work for x-axis
• PR #1193: Cairo backend ignores alpha in imshow.
• PR #2768: DOC/BUG: Fix references to demo files
• PR #2744: handle NaN case nicely in _is_sorted
• PR #2761: Fix line color handling
• PR #2763: double_pendulum_animated.py in 1.2.1 fails due to clear_temp kwarg
• PR #2756: Removes artificial limit in artist picker traversal. There are quite a
• PR #2555: Make it possible to add mpl.rcParams to itself or deepcopy
• PR #2558: fixes issue #2556
• PR #2762: BUG : makes Axes.margins work with just kwargs
• PR #2643: ENH/REF: Overhauled boxplots
• PR #2734: Fixed issue #1733 - AxesImage draw function now takes into account the
• PR #2757: Added missing warnings import
• PR #2753: BUG : fixes py3k import
• PR #1227: Does the gtk3agg backend work on python3?
• PR #2751: BUG : fix failing test on 3.2
• PR #2749: Qt4 keys
• PR #2137: PIL -> Pillow
• PR #2705: Build fails on OS X with NumPy 1.9
• PR #2707: Callable date formatter
• PR #1299: Update Axes3D.tricontour for custom triangulations
• PR #2474: MEP12: Example clean-up for reference
• PR #2727: Typo in explanation of annotation_demo
• PR #2728: fixed comment white space pep8
• PR #2720: Look for user-specified styles in ~/.config/matplotlib/stylelib
• PR #2712: Anchored sizebar fontprop
• PR #2713: Compare pep
• PR #2207: color of candlestick lines
• PR #2551: Fix behavior of hist function when passed empty dataset
• PR #2595: EHN: add a span_stays option to widget.SpanSelector
• PR #2647: use GridSpec in plt.subplots
• PR #2725: DOC : fixes small typos in matplotlib.dates docs
• PR #2714: Deprecated matplotlib.testing.image_util.
• PR #2691: Change LogFormatterExponent to consistently format negative exponents
• PR #2719: Package initialization made possible when executed in environments with...
• PR #2718: Added missing cleanup decorator import.
• PR #2248: axes_grid1: ImageGrid respect the aspect ratio of axes.
• PR #2481: datestr2num of year and month fails on 29th, 30th, and 31st of month
• PR #2423: Off-axes markers unnecessarily saved to PDF
• PR #2239: Update of mlab.pca - updated docstring, added saving the eigenvalues.
• PR #2711: Fixes issue #2525
• PR #2704: Bugfix for issue #1747. Allows removal of figure text artists.
• PR #2696: Fix Tk keyboard modifier masks on Windows
• PR #2690: Build failure on MacOS X 10.5.8 (PowerPC G5) with Python 3.3.3
• PR #2628: improved get_ticklabels kwarg
Matplotlib, Release 1.4.3

- PR #2634: address FuncAnimantion trying to take lengths of generators
- PR #2468: Add “sage” colors to colors.py
- PR #2521: Fix backend_svg.RendererSVG.draw_text to render urls
- PR #2703: Updating regex used to split sphinx version string.
- PR #2701: Fix FancyBboxPatch Typo
- PR #2700: Consistent grid sizes in streamplot.
- PR #2689: Disable offset box clipping by default.
- PR #2693: Use mpl.checkdep_ghostscript function to find ghostscript
- PR #2679: Make test_save_animation_smoketest actually run
- PR #2504: Using qhull for Delaunay triangulation
- PR #2683: Close a figure with a type long or uuid figure number
- PR #2677: Make sure self._idle is set to True in all cases
- PR #2597: BUG: Add subplot spec eq
- PR #2650: Lightsource shade method parameters for color range definition
- PR #2665: MacOSX backend supports 2x DPI images and MathTeX.
- PR #2680: Deprecate toolbarqt4agg
- PR #2685: Remove a redundant comparison that raises an exception in Python 3
- PR #2657: different fix for comparing sys.argv and unicode literals
- PR #2646: Fix Gtk3 crash when running inside of IPython
- PR #2661: NF - see axes.get_label() when clicking on Edit curves lines and axes pa...
- PR #2676: Fix typo in _axes.vlines doc-string
- PR #2569: Explicitly cast the input array to float before doing anything to it
- PR #2671: Deprecate IPython-related Sphinx extensions
- PR #2656: Use IPython’s copy of ipython_console_highlighting Sphinx ext, if available
- PR #2515: overloaded _make_twin_axes on LocateableAxesBase
- PR #2659: DOC: Remove redundant colormaps from examples
- PR #2636: “usepackage[russian]{babel}” does not work in matplotlib 1.3.1
- PR #2648: Update backend_webagg.py
- PR #2641: plot_date: Set the default fmt to ‘o’
- PR #2645: Add option to show/hide the source link in plotDirective
- PR #2644: Small typo in the license.
- PR #2461: New style format str

288 Chapter 6. Github stats
- PR #2503: Fix interactive mode detection
- PR #2640: Axes.plot: remove set_default_color_cycle from the docstring
- PR #2639: BUGFIX: ensure that number of classes is always of type INT in Colormap
- PR #2629: backend_qt4agg: remove redundant classes. Closes #1151.
- PR #2594: New layout for qt4 subplotlib + QMainWindow -> QDialog
- PR #2623: setupext: put pkg-config -I, -L, -l locations at the head of the list
- PR #2610: improve docstring and add test for rgb(<float>)
- PR #2618: Fix issue 1172
- PR #2619: slight tweak to mpl_example
- PR #2626: minor pep8 to fix failing master builds.
- PR #2606: embedding_webagg example: Download button does not work
- PR #2588: Refactor mechanism for saving files.
- PR #2615: Fixes issue #2482 and adds note in matplotlibrc.template
- PR #2613: BLD Fix build failure on Python 3.4b1 for Windows
- PR #2459: pep8 for backend_pdf.py
- PR #2409: Fix bugs related to bottom kwarg in step histograms
- PR #2549: Add methods to control theta position of r-ticklabels on polar plots
- PR #2567: more informative exceptions for empty/not-existing images in compare_images()
- PR #2603: Correcting bad string comparison in lin-log plot aspect verification
- PR #2561: multi-colored text example
- PR #2236: Add easy style sheet selection
- PR #2582: fix initialization of AnnotationBbox
- PR #2574: Add axes.titleweight as an rc param
- PR #2579: MultiCursor: make events connected during __init__ accessible (for later removal)
- PR #2591: Fix infinite recursion in units with ndarray subclasses.
- PR #2587: Make backend_pgf more flexible when saving to file-handles or streams (fix #1625).
- PR #2554: User Guide Structure
- PR #2571: This fixes the problem brought up in the mailing list with the recent spectrum improvements
- PR #2544: Fix 2542
- PR #2584: Fix typo in legend documentation
- PR #2401: adds rcParam axes.formatter.useoffset
- PR #2495: fixed an encoding bug when checking for gs version
- PR #2581: affineBase.__eq__ should not raise an exception when the other does not ...
- PR #2462: Path effects update
- PR #2562: Just some small tweaks to the recipes
- PR #2550: Using a single-shot timer with the Wx backend raises an AttributeError
- PR #2198: Fix compilation on Solaris
- PR #2553: removing items from the call to six.iteritems
- PR #2547: fix removed api change regarding spectral functions
- PR #2514: Mpl toolkit pep8
- PR #2522: Add additional spectrum-related plots and improve underlying structure
- PR #2535: Move external libraries to ‘extern’ directory - correction
- PR #2534: cast argv to unicode before testing
- PR #2531: Move external libraries to ‘extern’ directory
- PR #2526: Minor doc fixes
- PR #2523: Unicode issue in EPS output when using custom font
- PR #2479: Rastized background color
- PR #2512: Fix saving to in-memory file-like objects in Postscript backend
- PR #2472: Plots using twinx draw on top of figure frame
- PR #2485: ENH better error message when wrong cmap name.
- PR #2491: Re-enabled PEP8 test, closing #2443.
- PR #2502: Updated the docs of pyplot.gca.
- PR #2428: BUG: Fixed object type mismatch in SymLogNorm
- PR #2496: Adding a missing ‘b’ back into two ‘bbox_’ kwargs
- PR #2494: Update scatter_demo.py
- PR #2486: make pep8 test routine reusable for other projects
- PR #2480: Use Pillow always on Travis
- PR #2406: BUG: Fixed github stats retrieval
- PR #2441: Catch stderr as well as stdout
- PR #2415: Bug: alpha parameter was ignored when fill color is #000000
- PR #2300: would crash if get_home() returns None
- PR #2420: Refactor WebAgg so it can communicate over another web server
- PR #2447: BUG: Fix boxplots with manual confidence intervals passed as a numpy array
- PR #2453: PdfPages: add option to delete empty file when closed
- PR #2458: pep8 clean up
- PR #2156: [Sprint] scatter plots are (reportedly) too slow
- PR #2464: Rename C++ variables to avoid use of reserved identifiers
- PR #2470: ENH: use checkdep_ghostscript to determine ghostscript executable
- PR #2469: BUG: gswin64c.exe not detected on Windows
- PR #2476: Updated the position of a few of the text examples because they were overlapping and hard to read.
- PR #2379: Make matplotlib.test() print meaningful messages when baseline images are not installed
- PR #2418: AssertionError with quiver, quiverkey, and an additional patch.
- PR #2425: DOC: axis_off wrongfully appears as a parameter to Figure.add_subplot
- PR #2445: Declare Numpy as a setup dependency
- PR #2336: Added check in autoscale_None for completely masked pcolor plots.
- PR #2460: minor pep8 fix on every file
- PR #2457: Privatize Text.cached
- PR #2433: Handle Unicode font filenames correctly/Fix crashing MacOSX backend
- PR #2455: Gitignore update
- PR #2446: Don’t set use_2to3 unless we have to.
- PR #2449: Qt4 clear before draw
- PR #2435: Explicitly catch TypeError when doing pyparsing monkeypatch check
- PR #2440: Pdfpages pagecount convenience getter method
- PR #2437: Fix randomly failing tests
- PR #2099: Updated coding standards test to raise an exception containing the PEP8 failiures.
- PR #2439: Use six.string_types instead of basestring.
- PR #2436: Catch explicit exceptions when setting locale
- PR #2430: Document API change in hist
- PR #2416: Multipage pdf with statement
- PR #2427: DOC: Add axes_api to documentation after the refactoring
- PR #2271: docs: add webagg-backend
- PR #2417: Adding possibility to remove invisible lines and patches from relim
- PR #2242: DOC: Use monospace for –
- PR #2426: Remove dead rms computation
• PR #2421: docstring fix
• PR #2382: New stlye qt calls
• PR #2351: Annotation refactor
• PR #2408: backend_pgf: fix str/unicode comparison errors (v1.3.x)
• PR #2407: backend_pgf: fix str/unicode comparison errors
• PR #2404: Fix backend_ps.py
• PR #2399: TypeError occurs when self.button=None in MouseEvents
• PR #2402: support tight_bbox for pgf output, fixes #2342 (v1.3.x)
• PR #2391: support tight_bbox for pgf output, fixes #2342
• PR #2396: Try UNIXy and Windowsy ways of setting locale
• PR #2331: Make optional backends respect setup.cfg
• PR #2393: use six.move for cStringIO
• PR #2372: Fix step histogram endline
• PR #2390: Transparent reparams
• PR #2383: BUG: Fix IndexError: too many indices with numpy 1.8
• PR #2386: locale breaks test suite
• PR #2229: Matplotlib does not display hatching when rendering to pdf in fill_between
• PR #2371: Corrections to cbook.warn_deprecated calls().
• PR #2381: don’t install python-dateutil==2.1 on python 3.3
• PR #2380: check if pyparsing <= is broken instead of checking the version
• PR #2374: Doc fix typos
• PR #2368: Set locale for tests
• PR #2226: Stop relying on 2to3 and use six.py for compatibility instead
• PR #2335: make sure we only perform absolute imports on loading a backend
• PR #2192: Follow the PSF code of conduct
• PR #2367: Fix a number of long-failing tests
• PR #2363: [bug correction] trirefine is now independant of triangulation numbering
• PR #2357: Better axis limits when using shared axes and empty subplots
• PR #2358: Broken IPython notebook integration
• PR #2352: changed colorbar outline from a Line2D object to a Polygon object
• PR #2054: Ipython/Webagg integration
• PR #2301: Upload test result images to Amazon S3
• PR #2319: fix draw_idle reference in NavigationToolbar2
• PR #2345: texmanager font selection crashes
• PR #2307: font_manager.py UnicodeDecodeError when starting ipython –pylab
• PR #2306: Mollweide latitude grid
• PR #2325: BF: guard against broken PyQt import
• PR #2327: Bar demo2 improve
• PR #2332: Except AttributeError when checking for gtk3 backends
• PR #2340: Fix #2339: render math text when using path effects
• PR #2338: issues with pyparsing 1.5.7 and python 2.7
• PR #2334: Remove disabled code.
• PR #2344: Fixed the issue of pyplot tutorial missing the show() command
• PR #2308: Make pyplot.bar color kwarg less ambiguous
• PR #2333: Fix wrong syntax for assert
• PR #2326: BUG FIX for Pull Request #2275: Fix incorrect function calls
• PR #2328: Fix PySide compatibility
• PR #2316: Replace the obsolete wx.PySimpleApp
• PR #2317: fix the docstring for scale_docs
• PR #2110: Fix rc grid parameter inconsistency
• PR #2278: Can’t find 64-bit GhostScript on win64
• PR #2266: pyparsing version parsing error
• PR #2262: View acceptsFirstResponder (for key_press_events)
• PR #2147: Make nonposy=’clip’ default for log scale y-axes
• PR #1920: finance ochl->ohlc
• PR #2059: Pep8 on many tests
• PR #2275: Fix Qt4 figure editor color setting and getting
• PR #2279: Pyparsing
• PR #2290: Fix a recursion problem with masked arrays in get_converter
• PR #2285: Handle prop=None case in AnchoredText.__init__()
• PR #2291: ENH: use an artist’s update() method instead of the setp() function
• PR #2245: Adding a flush_events method to the MacOSX backend
• PR #2251: Remove deprecated code marked for deletion in v1.3
• PR #2280: PEP8 on tri module
• PR #2282: Extend search path for PyCXX headers
• PR #2283: Incorrect overriding of sys.stdout
• PR #2158: Changes to anchored_artists.AnchoredSizeBar
• PR #1939: GTK error, failed building on command line
• PR #2265: WebAgg favicon serving error in Python 3
• PR #2267: Mention six in what’s new and install docs
• PR #2261: WebAgg performance improvements
• PR #1547: qt4_editor/formlayout.py TypeError: float() argument must be a string or a number
• PR #2260: texmanager font family fix

Issues (1342):
• #3672: Python3 pep8 fixes
• #3355: Unneeded argument in get_linestyle
• #3558: Adds multiple histograms side-by-side example
• #3665: Remove usage of raw strides member in _backend_gdk.c
• #3309: Explicitly close read and write of Popen process (latex)
• #3488: pep8ify examples (part2)
• #3589: ENH: add to_grayscale() method to color maps
• #3662: Make all classes new-style.
• #3646: Remove PyCXX dependency for core extension modules
• #3664: [examples] pep8 fix e251 e27*
• #3294: fix typo in figlegend_demo.py
• #3666: remove print from test
• #3667: A bug in mpl_toolkits.mplot3d.axes3d
• #3638: MNT : slight refactoring of Gcf
• #3387: include PySide in qt4agg backend check
• #3597: BUG/TST : skip example pep8 if don’t know source path
• #3596: Pep8 tests fails when running python tests.py from base mpl dir.
• #3661: Numpy 1.6 fixes
• #3660: shading tests + numpy 1.6
• #3635: fix pep8 error classes e20[12] and e22[12] in examples
• #3653: Make ScalarMappable a new-style class.
• #3547: Don’t use deprecated numpy APIs
• #2092: Move to new Numpy API
• #3601: matplotlib.style.available not updated upon adding/deleting .mplstyle files
• #3616: matplotlib.pyplot.imread silently fails on uint16 images.
• #3628: Document auto-init behavior of colors.Normalize and cm.ScalarMappable.
• #3640: figure.max_num_figures was renamed to figure.max_open_warning.
• #3650: Typo fixes. [backport to doc branch]
• #3651: Error when saving rasterized figure to PDF
• #3470: MacOSX backend breaks for matplotlib 1.4 after importing seaborn
• #3564: Rcparam validation fix
• #3642: TST : know-fail shadding tests
• #3641: Annotations with Latex code cause errors in 1.5 master
• #3623: Qt5 backend doesn’t work with Qt 5.3
• #3636: mp4 is a container format, not a codec
• #3632: Fix for #3623
• #3639: Shading tests failing on master
• #3619: PatchCollection: pass other kwargs for match_original=True
• #3617: PatchCollection.__init__ ignores all kwargs if match_original=True
• #3637: typo: mp4 -> mpeg4. Closes #3636.
• #3629: examples: fix pep8 error class E211
• #2873: Add violin plots
• #3622: setup.py creates a zombie C extension called “freetype2”
• #3213: add whats_new entry for nbagg
• #3392: Cannot pickle figure or axes (TypeError: instancemethod)
• #3614: Pickling imshow fails (?)due to _imcache)
• #3606: nbagg issues with ipython 3.0
• #3494: corrupt eps output on python3
• #3627: Fixed Image and Renderer pickling
• #3515: examples: fix pep8 error classes E111 and E113
• #3625: animate_decay.py example code is less complicated
• #3621: matplotlib.__version__ is now unicode as of 1.4.0
• #3505: Interactive mode not working in 1.4
• #3620: Revert interactive determination
Matplotlib, Release 1.4.3

- #3311: Ship conda package metadata with matplotlib?
- #3248: Divide by zero error in matplotlib.tests.test_colors.test_light_source_shading_color_range
- #3613: Fix problem with legend if data has NaN's [backport to 1.4.x]
- #3618: UnicodeDecodeError when I try to import matplotlib from directory with non-ascii name
- #3313: BUG: 3 fixes for widgets (MultiCursor, SpanSelector, Slider)
- #3496: BUG: fix str vs bytes issue in py3 in ps backend
- #3609: Nbagg icons
- #3611: Fix spelling error
- #3607: Icon for font awesome 3.2.1 and 4 [backport to 1.4.x]
- #3605: matplotlib.pyplot.specgram generate bad image in 1.4.0
- #3604: regression in pandas test suite with mpl 1.4.0
- #3603: Error saving file (Qt5 backend)
- #2907: Expose ax.yaxis.labelpad and ax.xaxis.labelpad to the rc file
- #3096: Axes labelpad rc
- #3544: flier objects missing from structure return by boxplot
- #3600: BUG: now only set ‘marker’ and ‘color’ attribute of fliers in boxplots
- #3599: BUG: now only set ‘marker’ and ‘color’ attribute of fliers in boxplots
- #3516: import error when non-ascii characters are present in cwd or user name (windows)
- #3560: Updated whats new for nbagg and legend/patheffects docs.
- #3594: Unicode decode error [backport to 1.4.x]
- #3532: BUG: skip looking up expanduser if unicode error
- #3574: rotation parameter has no effect on RegularPolyCollection
- #3409: Win fixes
- #3573: Bug: Empty sym string for boxplots still displays fliers.
- #3595: Some small doc fixes only relevant on the master branch
- #3291: Lightsource enhancements
- #3459: boxplot in version 1.4.0 does not respect property settings for fliers (flierprops)
- #3592: Fix crash in picking for zero-length path collection
- #3590: Won’t use a font although it can be found by the FontManager
- #3591: BUG: Cannot pick empty vline
- #3586: Merge V1.4.x into master.
- #3585: merge V1.4.0-doc into v1.4.x
- #3412: Matplotlib 1.4 doesn’t install from source on CentOS 6
- #3423: Pytz should be specified and documented as a required dependency
- #3569: boxplot stats regression on empty data
- #3563: boxplot() and xticklabels
- #3566: BUG: don’t assume label in boxplot_stat
- #3567: Fixed the differencing of images for the webagg/nbagg backends.
- #3571: BUG: deal with empty list passed to boxplot
- #3533: BUG: fix handling of flierprop by boxplot
- #3581: pep8 cleanup on scatter_demo.py
- #3514: Ticks on top axis disappear if tick size is too large (when using bbox_inches='tight')
- #3578: Fixes test to assert instead of print
- #1713: Can’t store Unicode values in .matplotlibrc
- #3575: Supports locale-specified encoding for rcfile.
- #3479: Build dep updates
- #233: Make hist with ‘step’ histtype draw Line2D instead of Patch
- #3522: Inverting a datetime / plot_date y-axis
- #3570: matplotlib save dynamic user changes to plot
- #3568: Daily build fails at “import matplotlib.pyplot as plt”
- #3541: pyplot: Fix exception in _backend_selection during import [backport to 1.4.x]
- #3565: clabel randomly inconsistent when placed manually
- #3552: Nbagg enhancements
- #3559: Install texlive and other dependencies when building docs.
- #3550: Install texlive when building docs.
- #3555: Typo in comment documentation for example timers.py
- #3556: copy/paste corrections in test_backend_qt5
- #3545: Provide an informative error message if something goes wrong in setfont [backport to 1.4.x]
- #3551: Window isn’t drawn
- #3538: Importing matplotlib failing when package “six” is 1.3.0
- #3542: fix boxplot docs
- #3548: Silence some Sphinx warnings by rewriting docstrings.
- #3549: Texlive travis
- #3539: DEP: update six minimum version
• #3543: DOC : fix main-page tags
• #3524: Bug in AutoDateLocator when dates are in reverse order
• #3455: Documentation bug: boxplot docs have contradicting information
• #3468: boxplot() draws (min, max) whiskers after a zero-IQR input regardless of whis value
• #3436: matplotlib.use(‘nbagg ‘) does not work in Python 3
• #3464: BUG : nbagg py3k compatibility
• #3534: BUG : fixes whis over-writing in boxplot_stats
• #3529: Symlog norm still gives wrong result with integer lintresh.
• #3535: BUG/DOC : Correct default value listed in docstring
• #3537: 3D figures cannot be created in 1.4.0: ‘module’ object has no attribute ‘ _string_to_bool’
• #3369: Added legend.framealpha to rcParams, as mentioned in axes.legend docstring
• #3510: Fix setupext [backport to 1.4.x]
• #3530: Only insert links to pdfs if we are actually generating these.
• #3487: Can not import matplotlib when launching from non-ascii path
• #3527: Drawing an arrow using axis.annotate raises DeprecationWarning
• #3525: BUG : decode byte-strings in afm.py
• #3523: invalid EPS figure in Mac OS X
• #3526: BUG : fix eps corruption when using clipping
• #3492: Allow python 3 version of PyCXX
• #3521: More doc fixes
• #3504: postscript axes corner is not perfect
• #3520: a question about subplot in spyder
• #3513: examples: fully automated fixing of E30 pep8 errors
• #3512: What else apart from useoffset is controlling tick label offsets?
• #3461: This fixes a bunch of Sphinx warnings
• #3507: general pep8 fixes
• #3506: Named colors example, figure size correction [backport to 1.4.0-doc]
• #3493: Incorrect use of super() in mplot3d?
• #3439: Registering backends broken by backwards incompatible change
• #3511: Error in plot-gui while saving image
• #3509: Add Build Instructions for Windows 7 Using Visual Studio?
• #3503: Win fix simple
• #3495: BUG : don’t use super(self.__class__, self)
• #3500: Annotation xytext property does not return xyann value
• #3501: Bugfix for text.xytext property
• #3376: Move widget.{get,set}active to AxisWidget
• #3497: Ortho basemap projection with limits crashes
• #3447: cursor doesn’t change on keypress (GTKAgg backend)
• #3419: Better repr for Bboxes.
• #3474: call set cursor on zoom/pan toggle [backport to 1.4.x]
• #3425: Pep8ify examples
• #3477: Better check for required dependency libpng
• #3472: Memory leak displaying PIL image.
• #3484: TclError for draw_event handler calling close()
• #3480: Duplicate labels produced when using custom Locators/Formatters
• #3475: need for rubberband in zoom tool
• #3478: BUG : restore back-compatibility of register_backend
• #3471: Fix for invalid check of freetype version when no freetype library exists [backport to 1.4.x]
• #2900: Remove no-longer-necessary KnownFail for python 3.2.
• #3465: psd() draw a wrong line with sliced array(Matplotlib 1.4.0)
• #3454: backend_qt5 (1.4.0): Not saving the figure with NavigationToolbar (solved)
• #3467: Bugfix in mlab for strided views of np.arrays [backport to 1.4.x]
• #3469: Fix handling of getSaveFileName to be consistent [backport to 1.4.x]
• #3416: Specify difficulties installing mpl on OSX.
• #2970: add test of all the standard marker symbols
• #3318: Running setup.py egg_info starts to compile everything
• #3384: Test marker styles
• #3466: Invalid DISPLAY variable
• #3463: when executing a small script nothing happens!!
• #3462: Add legend.framealpha to matplotlibrc
• #2934: Line labels don’t update in the legend after changing them through the Qt4Agg dialog box
• #3431: Qt5 toolbar support not working in release 1.4.0
• #3407: Update dns/IP adress
• #3456: DOC : add known-bug + fix for QT5 toolbar issue
• #3458: DOC : minor import tweaks
• #3460: zoomed_inset_axes shows a incorrect result.
• #3457: Add Qt5Agg to backends in matplotlibrc.template.
• #3417: update citation page
• #3450: Wrong permissions when installing from source on Linux
• #3422: Use mailmap in github stats
• #3429: DOC : added pytz to optional dependencies docs
• #3434: Update tight_bbox.py [backport to 1.4.x]
• #3453: Bbox rebase 14
• #3452: Bbox fix rebase
• #3449: matplotlib/colors.py: modifying dict while iterating
• #3445: can’t bring plot to front eclipse after running the script on mac ox 10.9
• #3427: DOC : added DOI link to citing.html
• #3443: Pip install matplotlib does not work on Python 3.2 anymore
• #3440: V1.4.x (Mac backend’s “Save as ...”)
• #3438: Get rid of unused pre python 2.6 code in doc make.py
• #3414: Fixes TypeError when installing without freetype
• #2931: Added center and frame arguments for pie-charts
• #3411: fix rst mark up
• #3413: update freetype version in docs
• #3428: DOC : add caveat about freetype 2.3
• #3430: DOC : fixed markup / boxplot text in whats_new
• #3426: DOC : remove piwik code
• #3432: Update whats_new.rst
• #3396: Sort out OSX dmg files
• #2803: Get wx backends and examples comptible with Phoenix
• #3415: DOC: fix markup and download link in the Windows install section
• #3410: Latex rendering fails in ipython
• #3408: test_text Modify filter warning regex
• #3404: Wrong plot on basemap with lon=lat=1on=True
• #3406: A layer stacking problem of exported svg image compatible with inkscape
• #3282: Catch warning thrown in Mollweide projection.
• #2635: Crash on saving figure if text.usetex is True
• #3327: FontProperties are shared by all three titles of an Axes object
• #3328: copy FontProperty objects when creating titles (fix #3327)
• #3241: Cast to integer to get rid of numpy warning
• #3335: Fix and new test for #3327
• #3278: Use libpng-config if available
• #3244: Filter warnings in rcparams test (and others)
• #1980: Custom marker created from vertex list scales wrong
• #3378: BUG: Fixes custom path marker sizing for issue #1980
• #3395: Update Downloads page
• #2545: Some of Russian letters are not visible in EPS
• #3405: The memory taken up from the RAM pool by imshow
• #1717: Definitive docs for how to compile on Windows
• #3401: enlarge on windows build instructions slightly
• #3398: windows build comments with winbuild repository link
• #3399: Cherrypick doc changes to v1.4.0
• #3397: Install guide tweaks
• #2999: Update and clarify installation documentation
• #3394: DOC: add note about np.matrix and pandas objects
• #2138: pyplot.scatter not converting x and y to a 1-D sequence when the input is a 1xN matrix...
• #3388: Build support with dependencies for Windows
• #3144: Backend documentation
• #3379: syntax warning in qt5 with 1.4.0rc4
• #2451: _macosx.so crash in build using Xcode 5
• #3362: 3D line object loses its color cycle in a function animation
• #3390: Move stylelib directory to mpl-data
• #3385: Regression with cx_support in 1.4.0rc4
• #3389: request: more than two axes/spine on plot
• #3386: add PySide to interactive_bk list
• #3383: Tkinter backend finishes with segmentation fault
• #3289: Float division patches
• #2881: Focus stays in terminal on OS X and 1.3.1
• #3381: BUG : make qApp global before using it
• #2060: Fix tox.ini
• #3380: Remove residual mention of TODO file, which no longer exists.
• #166: RuntimeError: CGContextRef is NULL with draw_artist
• #169: csv2rec encoding support
• #311: Intelligent log labels
• #374: Add general rcParam mechanism for text
• #449: stem plots have no color cycling mechanisms
• #862: The y-axis label of figures created with psd() should not say “Density” when scale_by_freq=False
• #1021: Hatching Inconsistencies
• #1501: Panning and zooming does not work on axes created with twinx (and twiny)
• #1412: Path collection filling/stroking logic is different from the usual in the pdf backend
• #1746: pcolormesh with lambert projection ignores lower hemisphere
• #2684: Savefig to EPS with cyrillic title doesn’t work
• #1933: backend_pdf.py fails on 3d plots (1.3.x)
• #1996: Bug when installing in OS X with easy_install
• #2157: numpy/core/_methods.py:57: RuntimeWarning: invalid value encountered in double_scalars
• #2292: Axes label rotation
• #2343: Test failures
• #2448: idle_add deprecation warning.
• #2355: Type Error in bar3d plot when saved as svg
• #2361: pylab import fails for non-framework python installs on OS X
• #3349: DOC : added folders for api_changes and whats_new
• #2596: Latex formatting does not seem to work with xkcd style
• #2611: no __init__.py in matplotlib-1.3.1.win-amd64-py2.7.exe
• #2620: WebAgg for multiple clients
• #2686: Tornado error when using matplotlib WabAgg backend
• #2649: incorrect detection of text.latex.unicode=True
• #3367: macosx broken on python 3.4 non-framework builds, shaky on framework
• #3360: BUG : modified logic on starting qApp
• #3372: DOC: Fixed the wording of the deprecation warning
- #3366: feature request: set_data method for errorbar
- #3365: font configuration
- #3361: saving 3D line figure in pgf format results in error
- #3359: PEP8 conformity; removed outcommented code
- #3358: Don’t use unicode_literals in backend_qt4
- #3357: backend_qt5.py Don’t use six.u
- #3340: Plotting a dataframe from pandas: IndexError: list index out of range
- #3183: pep8ify examples/ part 3
- #3338: resizing figures in webagg
- #3353: Fix typo reported by @cel4
- #3240: Add documentation to Travis builds.
- #3287: DOC: comprehensive rewrite for OSX binary install
- #3348: fix deprecation warning in simple linear interpolation
- #3336: Boxplot shows wrong color for lower outliers
- #3337: BUG : don’t assign color to flier props if None
- #3245: First pass at the reorganization of content on the matplotlib home page.
- #3342: AGG link changed. Also fix other doc warnings
- #3262: 1.4.0 RC1: –ftversion vs –version freetype version
- #3214: add whats_new for webagg
- #3209: Install docs are hopelessly out of date
- #3322: Fixed error with QSizePolicy
- #3308: SVG backend font-weight bug
- #3344: Cairo backend math text
- #3339: Fix mathmpl images not showing in HTML Help (CHM)
- #3333: No response on editing axes by NavigationToolbar2 in interactive mode
- #3332: savefig crashes in backend_p[df][s].py when using plot-option mew
- #3331: Restore compatibility with Python 3.2
- #3304: 1.4.0 RC1+7: * glibc detected * python: corrupted double-linked list
- #3324: Fix #3304.
- #3325: ENH: add HTML Help builder option
- #3326: Docs build failure on Launchpad.
• #3329: Workaround for Sphinx not escaping _ (underscore) for tex output
• #3323: Replaced unicode() function by six.text_type
• #3321: SymLogNorm returns ‘inf’ and ‘nan’ when given negative vmin as __init__ argument
• #3223: get colorbar slides
• #3259: Attribute error when testing on system without ghostscript
• #3301: Colormap choice guidelines in documentation - based on talk at SciPy 2014
• #3320: DOC: Fix Malformed table. Text in column margin
• #3319: colorbar
• #3317: TST: Fix test_animation RuntimeErrors on Windows
• #3297: test_mplot3d.test_quiver3d tests require np.meshgrid from numpy >= 1.7.0
• #3310: Fix MatplotlibDeprecationWarning: The “loc” positional argument to legend is deprecated
• #3312: BUG: fix test error when ghostscript not installed
• #3299: 1.4.0 RC1 UserWarning: Rasterization of PolyCollection will be ignored
• #3307: SVG backend font-weight bug
• #3306: BUG : restore allow_rasterization on PolyCollection
• #3302: Check for GhostScript in this test which needs it
• #3194: Annotate bbox darrow
• #3277: MNT : better error handling on determining gs version
• #3300: Quiver3d fixes
• #3284: BUG : fix _reshape_2D bug with [(n, 1), ..] input
• #3220: pylab_examples/boxplot_demo.py crashes
• #3296: V1.4.x
• #3295: Use the interpolation parameter in make_thumbnail
• #3288: put generated .aux file into .gitignore
• #3283: Suppress invalid argument warnings in inverse Mollweide projection
• #3280: Docs build failure on Launchpad.
• #3235: Silence some more warnings
• #3281: Error with pip install with Python 3.4
• #3252: ImportError: No module named ‘mpl_toolkits’
• #3264: 1.4.0rc1: Python-level memory “leak” (internal font cache?)
• #3274: BUG : fixes FontProperties memory leak
• #3275: TST: Fix ImportError: No module named ‘mpl_toolkits’
• #3276: free type memory leak
• #3250: Fix WindowsError: [Error 32] The process cannot access the file
• #2918: re-write contribution guide lines
• #3170: DOC : whole bunch of documentation clean up
• #3115: do not recommend using pyplot in scripts
• #3271: Fix use of a non-integer index in split_path_inout.
• #3247: Usage FAQ
• #3255: Out of memory failures on Travis
• #3268: Travis broken
• #3270: DOC : windows install docs from cgholkhe
• #3257: MRG: refactor and bugfixes for plot_directive
• #3238: OSX install
• #3258: Fix various memory leaks discovered through valgrind
• #2908: 404 links on the screenshot page
• #3253: Stop make.py removing generated documentation figs
• #3269: Upload artifacts only on main repository.
• #3251: Add animation.convert_path setting to matplotlibrc.template
• #3266: remove obsolete TODO files
• #3261: Get rid of warning about GTK3Agg with python3
• #3260: webagg backend does not show figures due to JS error
• #3256: Fix memory leak in FT2Font.
• #3249: TST: Fix test_backend_ps failures on Windows
• #3254: Won’t write approx LaTeX character in legend?
• #3234: Put PEP8 tests in its own Travis configuration
• #3217: Added some function arguments to the documentation for FuncAnimation
• #3243: Fixed backend workflow.
• #3246: Fix some hyperlinks in the documentation
• #3004: FAQ and unit/ still refers to nxutils
• #2758: Test toolkits rebase
• #2533: Bug in mplot3D with PolyCollection: (0, 0) data point is always inserted into the data set.
• #2045: PolyCollection path closing is projected incorrectly by add_collection3d
• #2928: matplotlib.sphinxext.plot_directive.py issue with ..image:: directive option passing for latex output.
• #2975: webagg generated JS quotes
• #3198: Fix deprecation warning in simple_linear_interpolation
• #3152: OSX test failures
• #3175: Navigation toolbar, Save button, last used folder path
• #3197: Memory Leak in Agg
• #3239: Fix auto-closing in PolyCollection
• #3193: Fix plot directive when used with multiple options.
• #3236: Test PEP8 stuff in separate Travis build.
• #3188: Np error patch
• #3186: Numpy 1.9 issues.
• #3154: whitelist mpl_toolkits tests
• #3230: DOC : added note about useofset rcparam
• #3216: edit useofset docs in ticker to mention rcparam
• #3228: DOC : top_level doc-string clean up
• #3190: Adding two new styles to mplstyles
• #3215: Close files in animation to silence some warning in the test suite on python3
• #3237: Fix Collection3D. Fixes legend for scatter3d
• #3233: Update numpy version in setup.py
• #3227: Whats new cleaning
• #3224: Fix lots of warnings in docs/Examples that crash
• #3229: DEP : bump min numpy to 1.6
• #3226: bump numpy version to 1.6
• #3191: Test errors with numpy 1.5 - advice?
• #3222: add reduce to the list of imports from six.moves
• #3126: insertion of Annotation class docs into annotate docstring broken
• #3221: Fixes #3219 by ignoring pep8 noncomplicant auto-generated file.
• #3219: pep8 test failure on macosx
• #1541: Transparency of figures in 3D plots (mplot3d)
• #2227: Refactor of top-level doc/README.rst
• #3211: Mplot3d/depthshade
• #1692: switch to turn off auto-shading in scatter3D
• #3218: Started simple reorganization of the gallery examples
• #2487: WebAgg kills IPython kernel
• #3184: DOC : added warning to doc of get_window_extent
• #3055: Add warning to get_window_extent
• #3165: Bug restore boxplot defaults
• #3042: boxplot does not take parameters into account
• #3207: Fix memory leak in tostring_rgba_minimize(). (#3197)
• #3210: Fix PEP8 error.
• #3203: Make type1font.py work better on Python 3.x
• #3049: PDF Embedded fonts with python3 mpl reported as ‘Unknown’ by pdffonts and pdf readers
• #3155: BUG : fix fetch of freetype version during build
• #3192: TST : drop 3.2, add 3.4
• #3090: Set up travis to test 3.4/drop 3.2
• #3121: Added ‘PyQt4v2’ to valid values for backend.qt4
• #2977: RC backend.qt validation too limiting.
• #3167: BUG : raise exception in subplot if num out of range
• #3166: subplot(x, x, 0) should raise Exception
• #3208: Add missing import of unichr from six.
• #3156: DOC : added whats_new entry for Qt5 backend
• #2475: BUG: manual clabel positioning broke between 1.2 and 1.3
• #2843: BUGFIX: This change fixes #2475, where contour labels added manually
• #3205: BUG : patch from mikofski to fix webagg example
• #3204: embedded_webagg.py example needs patches
• #3202: dateutil isn’t included in 1.3.1
• #3199: triplot, etc examples broken by merged PR #3182
• #3201: Revert “[examples/api] autopep8 + use np.radians/np.degree where appropri... 
• #3200: Revert “pep8ify more examples in examples/ + use np.radians/np.degrees”
• #3174: MNT : replace and deprecated qt4Compat
• #3172: replace qt4Compat.py
• #3112: BUG : patches.Wedge.set_radius set wrong attribute
• #2952: BUG : turned clipping off on pie chart components
• #2518: pie chart is trimmed
• #2951: BUG/API : tweaked how AnchoredSizeBar handles font properties
• #2394: AnchoredSizeBar does not respect FontProperties size setting.
• #3157: BLD : fix build on windows
• #3140: Building issue under windows.
• #3189: BUG: use unittest.mock for Python 3.3+
• #3045: Use less aggressive garbage collection
• #3044: matplotlib shouldn’t call gc.collect()
• #3185: DOC : added details about r/cstride in plot3d
• #3143: Document r/c stride in plot_surface/wire frame
• #3182: pep8ify more examples in examples/ + use np.radians/np.degrees
• #3181: [examples/api] autopep8 + use np.radians/np.degree where appropriate
• #3163: DOC : documented bottom kwarg of hist
• #3136: bottom keyword argument of hist() not documented
• #3180: DOC: Fix order of parameters in ax.text docstring.
• #3178: Regression in IPython Sphinx extension
• #3176: rendering bugs in log/log-base-2 histograms
• #3168: DOC : add prominent doc about set_useOffset
• #2796: pyplot.plot casts integer tick values to floats
• #3171: Changing the legend fontsize “hides” dotted lines in the legend
• #3162: BLD : made tornado an optional external package
• #3039: tornado not optional
• #3169: Update pyplot_tutorial.rst
• #1026: Feature request: Quiver plot in Axes3D object
• #2268: _update_patch_transform(): AttributeError: ‘Rectangle’ object has no attribute ‘_y’
• #3084: Improving plt.hist documentation
• #1847: Crash when creating polar plot with log scale on radial axis
• #3160: Glade tutorial branch fixed
• #3008: Nbagg backend
• #3066: BUG: restore default boxplot style and respect for the sym kwarg
• #3164: fix bad pathing in whats_new.rst
• #3161: Docs build failure
• #3088: Add tutorial file for Glade 3.16-GTK3.10-Python 3
• #3159: BUG: fix qt4 backends
• #3158: backend_pgf: Error message for missing latex executable (fix #3051)
• #3051: improve error message when pgf can’t find tex executable
• #3125: DOC: added annotation example to arrow docstring
• #2350: Arrows affected by data transform
• #3149: 3dquiver rebranch
• #2961: Implementation of 3D quiver
• #3141: BUG: Fix ‘TypeError: expected bytes, str found’ on Python 3
• #3072: Implement backend for PyQt5 + modify Qt4 backends to use Qt5 module via shim
• #3153: Avoid floating point sensitivity in trisurf3d test
• #3151: document api changes
• #3139: savefig() saves different aspect ratio than show()
• #3138: ENH: Function to “reset” the color cycle on a set of axes
• #3145: Error in subplots sharey docs?
• #3147: Fix doc for sharey keyword in pyplot.subplots.
• #2958: feature request: set figure sizes w.r.t. screen resolution
• #3082: GTK-Glade tutorial is out of date
• #3133: Doc cleanup
• #3110: BUG: Add Figure.delcolorbar() to fully delete a colorbar
• #2688: Deleting axis in matplotlib > v1.2.1 does not work similar to v1.1.1
• #3131: DOC: sixify unichr
• #3117: Qt4 backend using unichr() in python3
• #3132: DOC: added note about maintain ref to widgets
• #3105: Sliders unresponsive when created inside a function
• #2927: BUG: don’t use mutable objects as dictionary keys
• #2828: PS backend fails to save polar plot
• #3122: DOC: mention Anaconda; clean some old junk out of the FAQ
• #3130: Scatter set sizes whats new
• #3113: BUG: PathCollection’ object has no attribute ‘set_sizes’
• #3127: DOC: added inherited-members to Axes autodoc
• #3128: Axes aspect doc
• #2608: Docs: pyplot.axes() should mention the aspect keyword argument
• #3103: errorbar: fmt kwarg defaults to None; use ‘none’ to suppress plot call
• #2366: Errorbar plot ignores linestyle rcParam
• #3123: DOC : add documentation to Polygon methods
• #3035: Add docs to Polygon *_xy
• #3124: Zooming to a point changes a picked point’s index for data longer than 100 points
• #3120: typo fix
• #2492: subplots() shared scale is off
• #3118: Wrong datalims with empty plots with shared axes
• #2963: Segmentation Fault on adding BBoxImage to matshow
• #3099: New animation example (Joy Division’s Unchained Love cover)
• #3111: bug fix: check the type of the ‘key’ of the two array ‘r1’ and ‘r2’
• #3093: Python 3.4 tkagg backend error while importing pyplot
• #3108: DOC : clarified doc of add_artist
• #3109: Undesired crop with thick lines
• #2822: DOC : added caveat to Axes.add_artist
• #2288: Symmetric Log scale: linscale < 1 ?
• #3106: small bug in ‘class Appender’
• #3107: Bug-fix for issue 3106
• #3079: Scatter plot color array length should raise Error
• #3092: Adds check that rgb sequence is of length 3
• #3100: Use autolim kwarg in add_collection to prevent duplication of effort.
• #3095: Memory issue when plotting large arrays with pcolormesh
• #2738: BUG : errorbar does not respect lines.linestyle rcparam
• #3104: BUG: in Spine.set_position(), preserve most Axis info.
• #2941: Order of ax.spines[].set_position() and ax.yaxis.set_major_formatter() produces different results
• #3012: set_ticks_position to non-default position, sets all tick texts to empty string
• #3097: scatter should take array for alpha
• #3101: Streamplot: clean up handling of masks, eliminate warning in test.
• #3102: Image: handle images with zero columns or rows.
• #3091: set_xlim() crashes kernel if interpolation=’none’
• #3098: Fix regression in pcolormesh: don’t generate a path list.
• #3094: Various improvements in finance.py
• #2929: clip_on documentation note/warning
• #3029: freetype cannot be found by build
• #3032: Fix unable to find Freefont on Ubuntu 14.04
• #3067: Fix for bug #3029.
• #3052: Unresponsive figure when using interactive mode on Windows
• #3078: fix argument checks in axis/base.margins
• #3086: Multiple test errors in current master on Python 3.4 / Ubuntu 12.04
• #2945: Bug in y-minimum for weighted, log, stepped Axes.hist
• #3089: Fix log hist y-axis minimum with weighted data
• #3087: small error in comment
• #2996: Violin Plots
• #3053: symlog-scale: Remove assert linscale >= 1.
• #3085: Mistake in documentation of Figure.colorbar()
• #3077: Invalidate font manager when rcParam family lists change.
• #3081: Points to pixels
• #2889: bug: path effects in text() change text properties
• #3080: Minor fix to commit 24bc071
• #3054: BUG : fix for text path rendering issues
• #3075: Add warning about updating font rcparams
• #3076: Bug: backend_pdf: UnicodeDecodeError: ‘ascii’ codec can’t decode byte 0xe2
• #3074: TST : force re-building of font-cache
• #3065: font priority bug
• #3073: TST : fix font-priority test
• #3047: Updated CHANGELOG and whats_new.rst: with (re-)generate auto-legend
• #2150: Bug in bar plot, leading zeros in data (bar heights) are ignored.
• #2942: #2150 Updating patch limits if either width or height is non zero: Issue 2150
• #2541: mouse-over coordinates wrong for polar plot with customized theta direction/offset
• #2874: Fix for issue #2541 (revised)
• #1981: plot() - Markevery only supports startpoint and stepsize, not endpoint
• #2662: allow slice and fancy indexing to only show some markers
• #2855: ENH Added the origin option to spy
• #3022: Updating PyQt version checks for v4.10+
• #3021: PyQt4 installation check fails as pyqtconfig is no longer built by default
• #2471: Qt5 Backend
• #3068: XDG_CONFIG_HOME causes server to crash
• #3010: How to set multiple default fonts with matplotlib?
• #3001: Install file got merge conflict
• #3033: Feature Request: Artists should have a name attribute?
• #3069: vistrails ImportError: No module named pylab.plot
• #3015: Date stem simplefix
• #2602: stem function with datetime argument does not work in 1.3.1
• #3000: PGF backend: Lines in multi-line text drawn at same position
• #3017: Do not provide (wrong) mtext instances for pre-layouted text blocks (fixes #3000)
• #3009: BUG: Showing a BboxImage can cause a segmentation fault
• #3061: Add Axes.add_image() for consistency.
• #3063: Change EPD links to Enthought Canopy
• #3050: Animation example: rain drops
• #1891: Animation module errors out when using Python3
• #1381: Figure.add_subplot documentation doesn’t explain args
• #2898: Fix animation errors
• #3031: avoid np.nan values in colors array returned by axes3d._shade_colors
• #3038: BUG : expand x/y range in hexbin if singular
• #2863: ensure non-singular extent in hexbin
• #3005: Remove all references to ‘ipython --pylab’
• #3048: Animation example displaying earthquakes
• #3043: CHANGELOG whats_new.rst / Re-generate simple auto legend
• #3040: OSX 10.7 Install Error
• #3018: Fix documentation of entropy function
• #2861: Update backend_cairo.py
• #3028: Import error QT4 backend with python3.2.3
• #3036: Unicode fixes
• #2871: Add a colorblind friendly heatmap.
• #2879: BLD : adjust min six version to 1.3
• #3037: DEP : removed levypdf from mlab
• #2974: documentation mistake in errorbar
• #3025: mpl issue: #2974 - documentation corrected
• #3030: Fix minor typo in customisation docs
• #3026: Bug in matplotlib.mlab.levypdf
• #3027: Update mlab.py
• #2947: Re-Generate legend, through apply_callback/Apply
• #3034: A function to refresh the figure.
• #3014: BUG : improved input clean up in Axes.{h|v}line
• #2197: pyplot.errorbar: problem with some shapes of the positional arguments
• #2771: Fix font family lookup calculation
• #3007: #3005 - Removed ‘ipython -pylab’ references
• #3003: PGF backend: Fix vertical positioning of multi-line text
• #2946: remove .rect member (clashes with QWidget)
• #2839: BUG : fixes non-float args to stem
• #2896: add doc for qt repaint
• #2837: EXP : turn of clipping in spine example
• #2772: BUG : instantiate fall-back writer
• #2651: in animation writer object not instanciated
• #2922: ENH : add flag to box_plot and bxp to manage (or not) xticks
• #2921: Boxplot resets x-axis limits and ticks
• #2950: DOC : edits to optional dependencies
• #2490: INSTALL should list ffmpeg/avconv/mencoder/imagemagick optional dependencies and versions
• #2916: Docs build segfaults on Launchpad
• #2995: Added ‘interpolation_none_vs_nearest’ example, without .DS_store files
• #3002: BUG/DOC : fix bad merge of INSTALL
• #2965: Feature Request: Data Cursor Mode
• #3006: The FreeType extension defines add_flags, but is never called. This
• #2899: adding linewidth argument to pie
• #2993: Avoid a null-pointer dereference in _tri.cpp
• #2994: Minor fixes in _macosx.m
• #2559: The response of mouse zoom & pan is slow with Qt4Agg backend.
• #2997: Disable copying of C++ classes with nontrivial destructors
• #2998: importing matplotlib breaks warn() function, when given an argument of type bytes
• #2992: Remove a few dead assignments
• #2991: Silence some compiler warnings related to ft2font
• #2969: Tarball not installing on mac osx 10.9.2
• #2990: Added ‘interpolation_none_vs_nearest’ example
• #2989: Don’t call Py_DECREF on null in _ttconv.cpp
• #2987: OpenCV + figure.show() doesn’t block GUI
• #2972: aliasing with imshow(z, interpolation = ‘none’), when saved as a pdf
• #2985: let KeyErrors pass silently on callback removal
• #2979: add ‘buttons’ parameter in LassoSelector and SpanSelector
• #2967: Exception with sphinx 1.2.2 using the ipython directive
• #2984: small error in install faq
• #2829: (fix #2097) PGF: get fonts from fc-list, use builtin fonts for tests
• #2097: PGF-related test failures on Mac OS-X
• #2976: Gtk3Agg backend (Ubuntu 14.04)
• #2982: Reset plot_directive context
• #2913: Allow :context: directive to take ‘reset’ option. Fixes #2892.
• #2914: Don’t close figure if context and apply_rcparams are both set.
• #2890: plot_apply_rcparams=True causes figure to not appear when updated
• #2983: DOC/BUG : fixed sphinx markup
• #2982: Docs build failure on Launchpad.
• #2981: TST: __spec__ (an import-related variable for modules) was added in pyth...
• #2978: BUG: EllipseCollection: fix transform error
• #2964: line style rendering error
• #2968: BUG: Fix the triangular marker rendering error.
• #2966: axvline doc typo fix
• #2962: py3k fix
• #2739: DOC : improved extent of imshow doc
• #2960: PEP8 : making pep8 happy again
• #2303: Document figure.get_size_inches, improve set_size_inches and improve a ValueError message
• #2836: DOC/ENH: get/set_size_inches
• #2953: col2hex in figureoptions.py not versatile enough
• #2956: Type of offset is int, not pointer.
• #2948: DOC: added missing doc changes from #2844
• #1204: Add power-law normalization
• #2452: Fixed issues with errorbar limits
• #2955: PEP8: add missing line to un-break build
• #2925: ‘Dictionary size changed during iteration’ in mathtext.py
• #2926: BUG: Removes iteration over locals (no-no) in mathtext
• #2915: Consistency of the radius argument for Path.points_in_path
• #2939: Fixes a bug in drawing bitmap images in the macosx backend for handling device scaling
• #2949: CLN: removed version check that required numpy > 1.2
• #2848: DOC: removed line about un-needed dependencies in Windows
• #2330: Documentation problem about installing matplotlib
• #2152: We don’t actually support Numpy v1.4
• #2943: Typos in doc of vlines
• #2944: Have pyplot.subplots return an np array no matter how many plots are created
• #2670: Core dump with use.tex
• #2938: Stem plots could handle dates, no?
• #2937: Distortion of vertical axis labels that contain MathTeX (MacOSX backend)
• #2940: Fix some documentation mistakes
• #2897: add ccw pie test
• #2933: #2897 Adding tests for pie ccw. Issue 2897
• #2936: #2897 Resubmitting the pull request for Issue 2897
• #2932: Py3K failure in transform.contains_branch
• #2923: Issue 2899
• #2930: Cranky pep8
• #2847: DOC: add link to plt.subplots from Figure.add_subplot
• #2919: Import hangs when importing pyplot
• #2909: BUG: add __hash__ to AffineBase
• #2924: Pyplot figure window container
• #966: axes_grid: indicate the axes for the suplot with append_axes
• #2906: Fix Cairo text on Python3 with pycairo
• #2903: Cairo Backend: Can’t convert ‘bytes’ object to str implicitly on Python3
• #2775: Compatibility with pandas 0.13
• #2920: fix six check message
• #2546: Candlestick shadow is drawn after candlestick body
• #2917: NotImplemented Error with gtk3cairo
• #2912: Fix paths in doc which are searched for matplotlibrc (XDG).
• #2735: Fixes issue #966: When appending the new axes, there is a bug where it
• #2911: text_axes missing cleanups
• #2905: Cairo: fix TypeError in figure.text() on Python3
• #2870: Wrong symbols from a TrueType font
• #2883: Fixing building issues on OSX 10.9 when using a non-system python installation
• #2902: Installer crash on Mac OS X 10.9.2 (crashed on child side of fork pre-exec)
• #2034: Legend tuple handler improve
• #2834: WebAgg: Fix IPython detection. Fix encoding error on Python 3
• #2853: counterclock parameter for pie
• #1664: Support for skewed transforms
• #2901: Trouble importing GTK when pyplot is imported
• #2844: BUG : Qt repaint workaround on windows
• #2895: typos: s/coodinate/coordinate & s/contols/controls
• #2894: typo: s/contols/controls
• #2891: wxversion
• #2601: Legend does not work for quiver
• #2875: Fix for issue #2872. Skip NaN’s in draw_path_collection.
• #2888: Is there any way to keep the length between ticks in symlog plot the same?
• #2887: fix a bug introduced in c998561d6cc1236
• #2882: [arm] segfault with matplotlib.mlab.PCA
• #2884: Fixed the failing tests on master.
• #2878: merging 1.3.x broke build on master
• #2159: Add darken and lighten to colors
• #2537: Clockwise pie diagram
• #2851: Fix positional/kwarg handling of the Z argument
• #2808: BUG: master has broken some 3d plots
• #2852: AttributeError: ‘module’ object has no attribute ‘next’
• #2876: Added called process error to __all__
• #2860: Fix subprocess.CalledProcessError on Google App Engine
• #2877: plt.xscale('log') overrides grid
• #2865: WebAgg: raise WebAggApplication.started flag before blocking
• #2872: Matplotlib “eats” points when zeros present on logscaled scatter plot
• #2867: GTK3 backend: implemented FigureCanvasBase.resize_event()
• #2868: backend_qt4 qt4_editor figureoptions get_icon crashes application
• #2866: ‘rounding’ of x coordinates in plt.plot with large 64-bit numbers
• #2864: Frame around colorbar doesn’t use closepath (in PDF renderer at least)
• #2862: how to realize the function like surf(x,y,z,c) in matlab
• #2858: BUG: colorbar autoscaling now ensures a finite range of values
• #2642: FloatingPointError exception in figure.colorbar
• #2859: BUG? subplot with sharex clears axes
• #2849: WebAgg issue - Uncaught SyntaxError: Unexpected token &
• #2856: Add labels to points to aid data exploration
• #2840: read Navigation toolbar parameters
• #2854: DOC hist is not cumulative by default
• #2825: WebAgg: extracted figure_div style into css and changed layout
• #2830: Bug in multiple step horizontal histograms
• #1455: Boxplot: allow whiskers to always cover entire range
• #2444: Fixed bad vector transforms.
• #2795: turn clipping off in spine example
• #2731: 2d padding
• #2846: Fix bug in horizontal step histograms (#2830)
• #2806: BUGFIX: contour.py - add_label_near()
• #2818: BUGFIX: This change fixes #2475, and incorporates pull request #2806
• #2824: WebAgg: drawing text is either skipped or duplicated
• #2776: TST : add time out to ani.save
• #2819: DOC: clarified docstring for cbook.boxplot_stats
• #2682: sphinx documentation, links, [I hate] orange
• #2835: quiver: handle autoscaling with quiverkey when animated
• #2616: Quiver does not _init with animated=True and quiverkey attached
• #2777: unicode strings u'' have leaked into test_legend.py
• #2769: squash smoke test on 3.2
• #2838: TST : make 3.2 pass again
• #2630: Qt4 save file dialog fails to appear on OSX
• #2347: Colorbar autoscale handling an array of one value
• #1499: twinx() on an inset axes wrongly acts on the main axes
• #2598: colorbar() TypeError: only length-1 arrays can be converted to Python scalars
• #2815: Bar plot width even for odd number of ‘left’ greater than 10
• #2832: WebAgg Python3 ... strings again
• #2833: path.simplify and path.simplify_threshold have no effect for SVG output
• #2826: GTK3 backend: Replaced deprecated GObject calls with GLib
• #2805: ENH: Updated inset locator axes to return a HostAxes by default
• #2652: Axis tickmarks of 1e20 and higher fail
• #2281: Documentation fails to build with Python 3
• #2807: Python 3 METH_VARARGS with METH_KEYWORDS
• #2202: Autoscale does not work for artists added with Axes.add_artist
• #2786: eventplot raises an exception for empty sequences
• #2821: DOC: point downloads at the matplotlib downloads
• #2817: Provide ‘lite’ version of release tar file
• #2164: [SPRINT] Single letter colors different than full name colors [sprint]
• #2809: BUGFIX: This change fixes #2475
• #2813: GTK3Agg backend: Only convert the cairo context to a cairocffi context o...
• #2810: Segfault when blitting multiple subplots with the gtk3agg backend
• #2814: nanovg backend?
• #2811: plot_surface displays darkened colormap
• #2801: Named colors example
• #2804: Fixed the image compare functionality.
• #2802: Getting Exception with “loc” attribute in title
• #1995: - wxPython Phoenix
• #2784: Scipy2013 Sprint: Cleaning F/C example
• #2798: Added remove methods for legends in figure and axes objects
• #2799: Xdg message repr
• #2781: Triplot returns the artist it adds.
• #2774: changed the text of INSTALL to be correct about what external
• #2788: MEP12: Clean-up line and marker demos
• #2787: Empty event loop
• #2792: Disable legend on matplotlib.axes instance
• #2779: remove old animation examples.
• #2027: Old animation examples
• #2791: Basemap background image has latitudes reversed
• #2789: Hatching color in contourf function.
• #2715: Upload packages to PyPI directly, for pip 1.5
• #2797: Memory black hole in matplotlib animation.
• #2794: fix typo in documentation
• #2793: missing mask for scroll event
• #2668: No “scroll_event” when using Gtk3 backends
• #2785: Log plots (semilogx, semilogy and loglog) crash with type error
• #409: Errorbar layering
• #2181: Scipy2013 Sprint: Cleaning examples of api example
• #2780: ENH : improve error invalid error message for subplot
• #2098: figure.add_subplot(1311): ValueError: Illegal argument(s) to subplot: (1, 3, 1, 1)
• #2228: Building docs: Could not import extension sphinxext.math_symbol_table (exception: No module named math_symbol_table)
• #2573: Matplotlib install breaks pip?
• #2373: python-dateutil encoding issues under python 3.3
• #2782: BUG: quiverkey must set the vector figure attribute
• #2389: table.py: fix issue when specifying both column header text and color
• #2729: update list of dependencies
• #2748: matplotlib 1.3.1 for Python 3.2.5 on Mac OS X produces corrupt .eps files
• #1962: When legend is outside the axes, pick events get handled twice
• #2755: Fixes legend.get_children() to actually return the real children of
• #1880: KeyEvent’s key attribute and modifier keys in WX backend
• #2599: Create interpolation_methods.py
• #2586: PGF backend does not clip image with specific bounding box
• #2621: Simplify and fix dpi handling in tight_bbox
• #2752: Make standardization of input optional in mlab.PCA
• #2773: matplotlib 1.3.1 is broken on windows
• #2557: Splitting navigation and toolbar
• #2726: Don’t snap pcolor
• #2760: line color='none’ regression in 1.3
• #2732: AttributeError: ‘Patch3DCollection’ object has no attribute ‘set_sizes’
• #2770: No way to pass clear_temp to Animation.save
• #2442: Rewrite of the entire legend documentation, including tidy ups of code and style to all things “legend”.
• #2747: Error with Savefig, PyParsing
• #2746: ENH : added warning on annotate
• #2675: clip_on = False does not work for x-axis
• #1193: Cairo backend ignores alpha in imshow.
• #2768: DOC/BUG: Fix references to demo files
• #2766: Docs build failure
• #1027: Possible bug in boxplot()
• #991: Perfectly horizontal or vertical lines don’t render to svg
• #841: Error autoscaling histogram with histtype='step'
• #217: New features for boxplot
• #2744: handle NaN case nicely in _is_sorted
• #2761: Fix line color handling
• #2763: double_pendulum_animated.py in 1.2.1 fails due to clear_temp kwarg
• #2765: There is no 1.2.2 out in the wild
• #2754: changed timeout on travis so that smoketest stops timing out
• #2756: Removes artificial limit in artist picker traversal. There are quite a
• #2543: rcsetup.validate_bool_maybe_none(None) raises Exception
• #2555: Make it possible to add mpl.rcParams to itself or deepcopy
• #2556: Quiver leaks memory when called multiple times
• #2558: fixes issue #2556
• #2767: Transparency of overlaid contour fill without effect on underlying isocontours
• #2740: MEP22 first draft (DO NOT MERGE)
• #2510: Axes.margins() raises ValueError when only **kwargs is used
• #2762: BUG: makes Axes.margins work with just kwargs
• #2590: wrong version of mpl_toolkits imported when installing mpl with python setup.py install --user
• #2730: ENH: added test on number of large differences between images
• #2643: ENH/REF: Overhauled boxplots
• #2669: PGF backend can’t find 64-bit ghostscript on win64
• #2540: Location of subplot.set_aspect(...) matters for imshow
• #2605: cxx error when installing matplotlib 1.3 on CentOS 5.9
• #2622: Matplotlib fails to build with Freetype 2.5.1 on OS X
• #2687: plt.xkcd() gives an error when a plt.text() is added with two line breaks “nn”
• #608: mpl_toolkits.axisartist should implement separate artists for x- and y- gridlines
• #2655: The “2to3” seems doesn’t work while building matplotlib1.3.1 with python3.x
• #1733: im.set_clip_path(rectangle) doesn’t work
• #2734: Fixed issue #1733 - AxesImage draw function now takes into account the
• #2757: Added missing warnings import
• #2135: Test toolkits rebase
• #2750: Jitter plot
• #1736: Implement a Colormap._reversed_
• #2753: BUG: fixes py3k import
• #2256: Can’t import plot_directive in Python 3
• #1227: Does the gtk3agg backend work on python3?
• #1030: patch facecolor does not respect alpha value
• #1703: matplotlib ignoring the switching of rendering backends
• #2710: Fixed empty text bbox drawing.
• #2751: BUG: fix failing test on 3.2
• #1429: [sphinxext] needs ability to build html without the link to source
• #1203: multi-subplot animation problem
• #2633: svg from filenames containing ‘—’ can be illformed
• #1148: Matplotlib doesn’t save correctly the figure when using patches.Circle on different plots
• #2264: Qt4Agg does not send backspace key_press_events
• #1947: Generate thumbnail of figure contents for use as figure window icon
• #2529: unable to build docs locally
• #2741: seg-fault building docs
• #1529: Unsatisfactory API example
• #2749: Qt4 keys
• #2137: PIL -> Pillow
• #2302: mpl_connect event.key has ‘alt’ prepended in matplotlib 1.2 on windows
• #2212: spyder and matplotlib
• #2273: Qt4 keys
• #2733: doc on rebase a pull request
• #2263: implemented support for asymmetric margins.
• #2192: QT backend: Post-plotting layout values set via GUI get lost after zoom-in/zoom-out cycle
• #1311: textcoords='axes fraction' does not work for some axes ranges
• #1712: Pylab function show() accepts any arguments
• #1567: Create kwarg to normalize histogram such that sum of bin values equals 1
• #2169: [SPRINT] Much better alpha control in mplot3d. Closes #1541
• #829: tight_layout: take suptitle into account?
• #2249: Autocompletion on rcParams: long-overdue restructuring of rcParams
• #2118: rc_file does not restore settings from my matplotlibrc
• #2737: Duplicate month name in AutoDateLocator on DST timezones
• #1408: Feature request: streaklines and improvements to streamplot
• #1060: AutoDateLocator.__init__: add version since which keywords are available
• #2237: Interactive plot styling
• #1413: Reminder: PySide decref patch
• #990: imshow extent keyword (documentation?)
• #379: Axes objects to hold dictionary of axis objects
• #2477: Add image value to x=, y= cursor text.
• #2483: animation with ‘ffmpeg’ backend incompatible with ‘bounding_box=tight’
• #2218: color should set both facecolor and edgecolor in pyplot.bar
• #2566: hsv_to_rgb isn’t the inverse of rgb_to_hsv
• #1561: mlab.psd returns incorrect frequency axis for two-sided spectra with nfft odd.
• #2365: Missing final edge in a ‘step’ histogram for matplotlib 1.3.0
• #2346: 2.7.5-r2: Fatal Python error: Segmentation fault at matplotlib/transforms.py”, line 2370 in get_matrix
• #2305: Request: Set figure size in pixels, not inches
• #2214: new figure invoked from a python shell in Emacs for win32 freezes console even after it’s closed
• #2235: Broken doc build
• #1901: Qt4Agg + PySide fails to open a plot on linux64 (CentOS-5,6)
• #1942: Matplotlib widgets: How to disconnect spanselector once selection is completed?
• #1952: import pylab; pylab.plot([1,3,2]): Failed to load platform plugin “xcb”
• #1863: SpanSelector broken in master
• #1586: frameon=False shifts plot axes to to the right and increases figure width
• #2121: geo_demo fails on OpenBSD
• #2091: PEP8 conformance test fails without listing location of failures
• #1738: Issue building on OSX 10.8.2
• #1080: patch for building with mingw32
• #867: Plots with many subplots can be slow
• #912: FreeSans horizontal misalignment in PDF, SVG, PS backends
• #1594: python3.3m/longintrepr.h:49: error: ‘PY_UINT32_T’ does not name a type
• #339: Use scrollbars when figure size is larger than screen
• #1520: “TextPath imported but not used”, says pyflakes
• #1614: Segfault ufunc_object.c:1750
• #1627: TkAgg backend: draw_if_interactive() broken?
• #1309: matplotlib.tests.test_mathtext.mathfont_cm_23_test.test makes python debug to crash
• #338: Interactive Compass object
• #1805: Each pyplot function deserves its own page
• #1371: vertical alignment of yticklabels fails on Θ
• #1363: error in matplotlib.pyplot.plot_date doku?
• #1308: plot_date should not use markers by default
• #1245: Cairo Backend: print_surface
• #224: Faster implementation of draw_rubberband in GTK+ backend
• #429: undefined behavior of figure.add_subplot() once subplot is modified.
• #2673: ‘key_press_event’ registering keypresses as alt-key combo (win8, python3.3, matplotlib 1.3.1)
• #2466: Implementation of separate horizontal/vertical axes padding to the axis_grid toolkit
• #2705: Build fails on OS X with NumPy 1.9
• #2707: Callable date formatter
• #1299: Update Axes3D.tricontour for custom triangulations
• #2007: mixed backend rely on __getattr__ method
• #2176: Adjustments to contour.py so that levels are set accordingly to input vmin and vmax, if pro-
vided, so color bar works correctly.
• #2474: MEP12: Example clean-up for reference
• #2727: Typo in explanation of annotation_demo
• #2728: fixed comment white space pep8
• #666: griddata constant spacing check needs tweaking
• #2313: mlab.griddata() more robust check for constant spacing
• #2720: Look for user-specified styles in ~/.config/matplotlib/stylelib
• #2712: Anchored sizebar fontprop
• #2713: Compare pep
• #2717: Unexpected behaviour in errorbar
• #2207: color of candlestick lines
• #2551: Fix behavior of hist function when passed empty dataset
• #2595: EHN: add a span_stays option to widget.SpanSelector
• #2647: use GridSpec in plt.subplots
• #2724: Documentation for WeekdayLocator.byweekday parameter incorrect?
• #2725: DOC : fixes small typos in matplotlib.dates docs
• #2714: Deprecated matplotlib.testing.image_util.
• #2691: Change LogFormatterExponent to consistently format negative exponents
• #2719: Package initialization made possible when executed in environments with...
• #2723: Backend selection without $DISPLAY available
• #2718: Added missing cleanup decorator import.
• #2695: Matplotlib useable on clusters with read-only resp. non-existing home directories on compute nodes.
• #1907: Timers independent of canvases
• #1780: New colormap normalizations: sqrt, arcsinh
• #1519: new methods set_xlabel and set_ylabel in “class Figure”
• #1254: backend_cairo.py – added print_surface method
• #1200: Hatch linewidth for pdf and ps backends (#235).
• #1143: [Sprint] Matlab fplot
• #1134: Enable mlab.rec2txt() to write small floats instead of just zero
• #827: make hist histtype='step' return list of container instead list of list
• #701: modify pcolor so masked elements are plotted with color specified in cmap.set_bad
• #801: tight_labels
• #2248: axes_grid1: ImageGrid respect the aspect ratio of axes.
• #2585: Improvements to anchoredArtists.AnchoredSizeBar
• #2481: dates2num of year and month fails on 29th, 30th, and 31st of month
• #2423: Off-axes markers unnecessarily saved to PDF
• #2500: WIP: Fix streamplot set
• #2239: Update of mlab.pca - updated docstring, added saving the eigenvalues.
• #2592: api docs for matplotlib.artist.get and matplotlib.artist.getp are exactly the same
• #1747: NotImplementedError: cannot remove artist
• #2525: cfl() doesn’t clear gcf()._suptitle
• #2711: Fixes issue #2525
• #2704: Bugfix for issue #1747. Allows removal of figure text artists.
• #2696: Fix Tk keyboard modifier masks on Windows
• #2709: matplotlib colors darker than equivalent matlab colors
• #2690: Build failure on MacOS X 10.5.8 (PowerPC G5) with Python 3.3.3
• #2507: AutoDateFormatter: More customizable formatting
• #2628: improved get_ticklabels kwarg
• #1715: axes.get_xticklabels() doesn’t return all tick labels.
• #2634: address FuncAnimation trying to take lengths of generators
• #1769: FunctionAnimator tries to take length of iterator
• #2468: Add “sage” colors to colors.py
• #2520: Adds option to plot average in boxplot, besides the median
• #2432: Rotate markers in Scatter plot
• #2521: Fix backend_svg.RendererSVG.draw_text to render urls
• #2703: Updating regex used to split sphinx version string.
• #2701: Fix FancyBboxPatch Typo
• #2653: Inconsistent streamplot grid size
• #2700: Consistent grid sizes in streamplot.
• #2689: Disable offset box clipping by default.
• #2530: AnchoredOffsetBox not taken into account by bbox_inches=’tight’
• #2693: Use mpl.checkdep_ghostscript function to find ghostscript
• #2692: _get_xdg_config_dir(), _get_xdg_cache_dir() - Restored for conformity to the standard.
• #1809: Delaunay bug: bad triangulations (intersecting triangles)
• #2679: Make test_save_animation_smoketest actually run
• #2660: Error with _compute_convex_hull on certain triangulations.
• #2504: Using qhull for Delaunay triangulation
• #2683: Close a figure with a type long or uuid figure number
• #2583: Pylab figure becomes unresponsive after an error
• #2677: Make sure self._idle is set to True in all cases
• #2597: BUG: Add subplot spec eq
• #2650: Lightsource shade method parameters for color range definition
• #2665: MacOSX backend supports 2x DPI images and MathTeX.
• #1958: Macosx: Retina displays are not supported
• #2112: New subplottool and various fixes
• #2680: Deprecate toolbarqt4agg
• #2685: Remove a redundant comparison that raises an exception in Python 3
• #2657: different fix for comparing sys.argv and unicode literals
• #2646: Fix Gtk3 crash when running inside of IPython
• #2661: NF - see axes.get_label() when clicking on Edit curves lines and axes pa...
• #2681: Plotting a matrix fails with maximum recursion depth exceeded.
• #2676: Fix typo in _axes.vlines doc-string
• #2569: Explicitly cast the input array to float before doing anything to it
• #2607: Allow global customization of ticker params
• #2672: Exporting 3d plots as u3d files
• #2671: Deprecate IPython-related Sphinx extensions
• #2674: properties instead of set_ and get_
• #2666: Deprecate IPython-related Sphinx extensions
• #2656: Use IPython’s copy of ipython_console_highlighting Sphinx ext, if available
• #2515: overloaded_make_twin_axes on LocateableAxesBase
• #2659: DOC: Remove redundant colormaps from examples
• #2658: “float() argument must be a string or a number” when saving a png
• #2636: “usepackage[russian]{babel}” does not work in matplotlib 1.3.1
• #2648: Update backend_webagg.py
• #2641: plot_date: Set the default fmt to ‘o’
• #2593: I’d like to see axes.get_label() when clicking on ‘Edit curves lines and axes parameters’ after plt.show()
• #2645: Add option to show/hide the source link in plot_directive
• #2644: Small typo in the license.
• #532: Figure.tight_layout() error or doesn’t work on Win with wxPython
• #1732: Explicitely passing an axes to share axis with
• #2461: New style format str
• #2503: Fix interactive mode detection
• #2640: Axes.plot: remove set_default_color_cycle from the docstring
• #2639: BUGFIX: ensure that number of classes is always of type INT in Colormap
• #2638: TypeError: Cannot cast scalar from dtype(‘float64’) to dtype(‘int64’) according to the rule ‘same_kind’
• #2629: backend_qt4agg: remove redundant classes. Closes #1151.
• #1151: FigureManagerQT used in new_figure_manager of Qt4Agg backend
• #2631: BUG: Raise exception for invalid input
• #2594: New layout for qt4 subplottool + QMainWindow -> QDialog
• #2623: setupext: put pkg-config -I, -L, -l locations at the head of the list
• #1451: 3D animation example no longer works.
• #1172: Axes.tick_params() fails with labelsize=<string> and direction='out'
• #2610: improve docstring and add test fot to_rgb(<float>)
• #2609: to_rgb(float) or to_rgb(str(flot))
• #2618: Fix issue 1172
• #2619: slight tweak to mpl_example
• #2626: minor pep8 to fix failing master builds.
• #2482: animation fails to create a movie with ‘ffmpeg_file’ backend
• #2443: Fix PEP8 test failures on master
• #2604: adding global variable to keep track if gtkmain has been called
• #2617: factor the pyplot dependence out of the figure_manager classes
• #1545: gtk backend should switch to gtk.Builder
• #2606: embedding_webagg example: Download button does not work
• #1646: Interactive mode broken in Qt4Agg backend?
• #1745: hist again... normed=True, stacked=True doesn’t make sense
• #2588: Refactor mechanism for saving files.
• #1196: errorbar bars don’t respect zorder
• #2412: FAIL: matplotlib.tests.test_axes.test_single_point.test
• #2411: FAIL: matplotlib.tests.test_axes.test_symlog2.test
• #2410: matplotlib.tests.test_image.test_rasterize_dpi.test failure
• #2329: set_position on Annotation is not working
• #1486: ENH: Boxplot median and confidence intervals computed with user-spec’d function
• #1181: NF - Text numeric values for ha and va
• #2614: Initialization fails if get_home() returns None
• #2615: Fixes issue #2482 and adds note in matplotlibrc.template
• #2493: Solve issue #2482
• #2613: BLD Fix build failure on Python 3.4b1 for Windows
• #2459: pep8 for backend_pdf.py
• #2612: Traitlets
• #2409: Fix bugs related to bottom kwarg in step histograms
• #2549: Add methods to control theta position of r-ticklabels on polar plots
• #2567: more informative exceptions for empty/not-existing images in compare_images()
• #2473: black background on rasterized quadmesh in ps output
• #2603: Correcting bad string comparison in lin-log plot aspect verification
• #697: Partial coloring of text
• #2561: multi-colored text example
• #2236: Add easy style sheet selection
• #2600: register Axes3D along with other projections
• #2582: fix initialization of AnnotationBbox
• #2574: Add axes.titleweight as an rc param
• #2579: MultiCursor: make events connected during __init__ accessible (for later removal)
• #2591: Fix infinite recursion in units with ndarray subclasses.
• #1625: saving pgf to a stream is not supported
• #2587: Make backend_pgf more flexible when saving to file-handles or streams (fix #1625).
• #2554: User Guide Structure
• #2565: mlab.psd behavior change
• #2571: This fixes the problem brought up in the mailing list with the recent spectrum improvements
• #2589: mathtext rendered doesn’t work with bundled pyparsing.py module
• #2544: Fix 2542
• #2542: Visual glitch in Axes borders
• #2584: Fix typo in legend documentation
• #2578: pie_chart_with_horizontal_bar_chart added
• #2400: Feature request: rc parameter for ‘useOffset=False’
• #461: ScalarFormatter creates useless offsets by default
• #2576: ENH: Change default value of useOffset to false in ScalarFormatter.
• #2401: adds rcParam axes.formatter.useoffset
• #2495: fixed an encoding bug when checking for gs version
• #2581: AffineBase.__eq__ should not raise an exception when the other does not ...
• #2572: PPA for Precise
• #2564: Axes3D scatter changes the color in version 1.2.1 during rotation
• #2462: Path effects update
• #2570: matplotlib 1.3.0 doc build for mac osx 10.9
• #2364: No official build of OSX version on the download page.
• #2563: Cannot hide axes ticks with log-scales
• #2562: Just some small tweaks to the recipes
• #2550: Using a single-shot timer with the Wx backend raises an AttributeError
• #2198: Fix compilation on Solaris
• #2552: after use('Agg'), the animate does not work well
• #2553: removing items from the call to six.iteritems
• #2547: fix removed api change regarding spectral functions
• #2514: Mpl toolkit pep8
• #883: bbox_inches=”tight” causes huge figures and text far outside figure frame
• #2548: Zoom/pan shifts displayed surface.
• #2538: streamplot hangs in application embedding Python interpreter
• #2522: Add additional spectrum-related plots and improve underlying structure
• #2535: Move external libraries to ‘extern’ directory - correction
• #2534: cast argv to unicode before testing
• #2531: Move external libraries to ‘extern’ directory
• #2526: Minor doc fixes
• #2523: Unicode issue in EPS output when using custom font
• #2513: Patch disconnected when moved to another axes.
• #2479: Rastized background color
• #2512: Fix saving to in-memory file-like objects in Postscript backend
• #2472: Plots using twinx draw on top of figure frame
• #842: Patch.update_from does not preserve the facecolor when alpha is set.
• #792: Make tests pass under all freetype versions
• #2498: savefig() with StringIO.StringIO() object failing for PS backend on Python 2.7.5
• #2252: Transparent SVGs not rendered correctly in PDF with ipython nbconvert
• #2485: ENH better error message when wrong cmap name.
• #2505: [Wishlist] fontproperties of table
• #2489: tiny correction of matplotlibrc.template
• #2506: minor docstring fix in dates.py
• #2491: Re-enabled PEP8 test, closing #2443.
• #2501: ttfFontProperty fails with invalid/misconfigured fonts
• #2502: Updated the docs of pyplot.gca.
• #2428: BUG: Fixed object type mismatch in SymLogNorm
• #2497: Handle floats with commas in AFM files
• #2496: Adding a missing ‘b’ back into two ‘bbox_‘ kwargs
• #2494: Update scatter_demo.py
• #2484: Fixes issue #2482 and recasts xrange as iter
• #2486: make pep8 test routine reusable for other projects
• #2480: Use Pillow always on Travis
• #2406: BUG: Fixed github stats retrieval
• #2441: Catch stderr as well as stdout
• #2415: Bug: alpha parameter was ignored when fill color is #000000
• #2300: would crash if get_home() returns None
• #2420: Refactor WebAgg so it can communicate over another web server
• #2447: BUG: Fix boxplots with manual confidence intervals passed as a numpy array
• #2453: PdfPages: add option to delete empty file when closed
• #2458: pep8 clean up
• #2463: _tri breaks build on Cygwin
• #2156: [Sprint] scatter plots are (reportedly) too slow
• #1814: ipython and matplotlib
• #2464: Rename C++ variables to avoid use of reserved identifiers
• #2470: ENH: use checkdep_ghostscript to determine ghostscript executable
• #2469: BUG: gswin64c.exe not detected on Windows
• #2476: Updated the position of a few of the text examples because they were overlapping and hard to read.
• #2379: Make matplotlib.test() print meaningful messages when baseline images are not installed
• #2293: 1.3.0: type of hist return value changed
• #2418: AssertionError with quiver, quiverkey, and an additional patch.
• #2425: DOC: axis_off wrongfully appears as a parameter to Figure.add_subplot
• #2196: 1.3.0rc4: FAIL: matplotlib.tests.test_image.test_rasterize_dpi.test
• #2445: Declare Numpy as a setup dependency
• #2336: Added check in autoscale_None for completely masked pcolor plots.
• #2377: ConnectionPatch “axis fraction” failure when axesB contains a plot
• #2460: minor pep8 fix on every file
• #2457: Privatize Text.cached
• #2433: Handle Unicode font filenames correctly/Fix crashing MacOSX backend
• #2455: Gitignore update
• #2454: AxesImage.set_cmap() and AxesImage.set_clim() have no effects.
• #2446: Don’t set use_2to3 unless we have to.
• #2449: Qt4 clear before draw
• #2424: remove some redundancy in _BaseAxes
• #2435: Explicitly catch TypeError when doing pyparsing monkeypatch check
• #2440: Pdfpages pagecount convenience getter method
• #2437: Fix randomly failing tests
• #2099: Updated coding standards test to raise an exception containing the PEP8 failiures.
• #2439: Use six.string_types instead of basestring.
• #2438: Use six.string_types instead of basestring.
• #2289: speedup figure rendering removal of .startswith() calls and use generato...
• #2431: BUG: in font_manager, handle unicode output from fontconfig
• #2436: Catch explicit exceptions when setting locale
• #2430: Document API change in hist
• #2311: Use mimetypes module rather than try to implement our own
• #2416: Multipage pdf with statement
• #2427: DOC: Add axes_api to documentation after the refactoring
• #2271: docs: add webagg-backend
• #2417: Adding possibility to remove invisible lines and patches from relim
• #2429: Document API change in hist
• #947: Explain @cleanup decorator in testing documentation
• #2096: Drawing a histplot crashes deeply inside matplotlib
• #2242: DOC:Use monospace for –
• #2426: Remove dead rms computation
• #2421: docstring fix
• #2419: extra ticklocs and ticklabels when plotting with bar(log=True) in matplotlib >= 1.3
• #2378: issues with unicode_literals and pysides.QtCore.Slot
• #2414: Adding the possibility to exclude invisible lines and patches in relim
• #2382: New stlye qt calls
• #2351: Annotation refactor
• #2297: Fix bugs related to bottom kwarg in step histograms
• #2117: fixes #2097, pgf: get scalable system-fonts from fontconfig, rely on builtin fonts for tests
• #2408: backend_pgf: fix str/unicode comparison errors (v1.3.x)
• #2407: backend_pgf: fix str/unicode comparison errors
• #2403: PGF backend tests failing
• #2404: Fix backend_ps.py
• #2405: Operator ‘<<’ is deprecated, use ‘<<=' instead
• #2399: TypeError occurs when self.button=None in MouseEvents
• #2402: support tight_bbox for pgf output, fixes #2342 (v1.3.x)
• #2391: support tight_bbox for pgf output, fixes #2342
• #2342: savefig PGF - RuntimeError: Cannot get window extent w/o renderer
• #2398: printing MouseEvents throws TypeError when button==None
• #2397: request : could the default windows font of matplotlib support an utf-8 “µ”?
• #2396: Try UNIXy and Windowsy ways of setting locale
• #2392: PDF link for documentation of version 1.3.0 gets documentation for version 1.2.1
• #2331: Make optional backends respect setup.cfg
• #2393: use six.move for cStringIO
• #2372: Fix step histogram endline
• #2395: Drawing arrows correctly in log scale plots is too hard
• #2385: fix forgotten cStringIO replacement when replacing 2to3 with six
• #2384: Make background transparent by default when saving figure
• #2388: Set default for transparency savefig
• #2390: Transparent rcparams
• #2383: BUG: Fix IndexError: too many indices with numpy 1.8
• #2376: feature test pyparsing 2 bug instead of version check
• #2386: locale breaks test suite
• #2229: Matplotlib does not display hatching when rendering to pdf in fill_between
• #2371: Corrections to cbook.warn_deprecated calls().
• #2381: don’t install python-dateutil==2.1 on python 3.3
• #2380: check if pyparsing <= is broken instead of checking the version
• #2374: Doc fix typos
• #788: font_styles test failing for some people
• #1115: Pass text alignment information to the PDF, PGF and PS backends
• #1121: Cairo text is misaligned vertically
• #1157: Use automatic dependency installation
• #1342: Make test data install optional
• #1658: WebAgg backend blocks
• #2021: Running tests in parallel occasionally hangs in unpredictable ways
• #2062: Deprecate our IPython-related Sphinx directives
• #2286: Don’t enable “interactive mode” if not at the console
• #2368: Set locale for tests
• #2226: Stop relying on 2to3 and use six.py for compatibility instead
• #2320: fail to import matplotlib.pyplot
• #2310: Fix wrong binding of JS this
• #2335: make sure we only perform absolute imports on loading a backend
• #2309: use(‘module://’) directive doesn’t work as expected
• #2192: Follow the PSF code of conduct
• #2367: Fix a number of long-failing tests
• #2363: [bug correction] trirefine is now independant of triangulation numbering
• #2356: Bad xlim/ylim when using shared axes subplots and an empty subplot
• #2357: Better axis limits when using shared axes and empty subplots
• #2362: Non-ascii font name makes matplotlib.pyplot fail at import
• #2358: Broken IPython notebook integration
• #2296: rc defaults incorrectly interpreted by colorbar
• #2352: changed colorbar outline from a Line2D object to a Polygon object
• #2354: Possible memory leaks reported by valgrind
• #2054: Ipython/Webagg integration
• #2191: Write a dependency policy
• #2301: Upload test result images to Amazon S3
• #2348: Ignore any non-ascii characters when reading font information.
• #2319: fix draw_idle reference in NavigationToolbar2
• #2349: Plot errorbars as boxes instead of bars
• #2345: texmanager font selection crashes
• #2307: font_manager.py UnicodeDecodeError when starting ipython –pylab
• #2306: Mollweide latitude grid
• #2299: Mollweide projection no longer shows horizontal gridlines
• #1648: RuntimeError: No SFNT name table
• #2325: BF: guard against broken PyQt import
• #2327: Bar demo2 improve
• #2332: Except AttributeError when checking for gtk3 backends
• #2134: MatPlotLib Figure Freezing on Windows
• #2340: Fix #2339: render math text when using path effects
• #2338: issues with pyparsing 1.5.7 and python 2.7
Matplotlib, Release 1.4.3

- #2334: Remove disabled code.
- #2344: Fixed the issue of pyplot tutorial missing the show() command
- #2339: Math text with path effects is rendered as plain text
- #2337: pyplot tutorial’s codes are missing statements
- #2308: Make pyplot.bar color kwarg less ambiguous
- #2333: Fix wrong syntax for assert
- #2321: PDF backend failure
- #2318: clean up of Qt namespace breaks PySides
- #2326: BUG FIX for Pull Request #2275: Fix incorrect function calls
- #2328: Fix PySide compatibility
- #2323: Axes cannot be animated using animation.py with blit
- #2322: Axes cannot be animated using animation.py with blit
- #2312: Replace the obsolete wx.PySimpleApp in the demo file embedding_in_wx5
- #2316: Replace the obsolete wx.PySimpleApp
- #2317: fix the docstring for scale_docs
- #2244: Upgrade requirement to pyparsing 2.0.1, and fix pyparsing deprecation warnings
- #2109: rcparam['axes.grid']=True != axes.grid(True) ?
- #2110: Fix rc grid parameter inconsistency
- #197: FigureCanvasMac.flush_events() raises NotImplementedException
- #2278: Can’t find 64-bit GhostScript on win64
- #2266: pyparsing version parsing error
- #2262: View acceptsFirstResponder (for key_press_events)
- #2255: XKCD-style doesn’t work with LineCollection
- #949: add AOSA chapter link to docs?
- #2147: Make nonposy='clip’ default for log scale y-axes
- #163: Problem with errorbar in log scale
- #1920: finance ochl->ohlc
- #2059: Pep8 on many tests
- #2295: Vertical text alignment in multi-line legend entries
- #2275: Fix Qt4 figure editor color setting and getting
- #2274: Figure editor incorrectly captures color properties
- #2287: Alpha values smaller than 1/256
• #2279: Pyparsing
• #2290: Fix a recursion problem with masked arrays in get_converter
• #2285: Handle prop=\texttt{None} case in AnchoredText.__init__()
• #2291: ENH: use an artist’s update() method instead of the setp() function
• #2245: Adding a flush_events method to the MacOSX backend
• #2251: Remove deprecated code marked for deletion in v1.3
• #2280: PEP8 on tri module
• #2282: Extend search path for PyCXX headers
• #2283: Incorrect overriding of \texttt{sys.stdout}
• #2158: Changes to anchored_artists.AnchoredSizeBar
• #2284: plt.hist(... histtype='step') draws one line too much
• #1939: GTK error, failed building on command line
• #2272: zombie webpages
• #2269: examples/showcase/xkcd.py does not show line randomization on Mac OS X
• #2265: WebAgg favicon serving error in Python 3
• #2267: Mention six in what’s new and install docs
• #2261: WebAgg performance improvements
• #1547: qt4_editor/formlayout.py TypeError: float() argument must be a string or a number
• #2206: 1.3.0rc4: FAIL: No such file or directory: u’.../doc/mpl_toolkits/axes_grid/examples/demo_floating_axis.py’
• #2260: texmanager font family fix
Matplotlib only uses BSD compatible code, and its license is based on the PSF license. See the Open Source Initiative licenses page for details on individual licenses. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits. For a discussion of the motivations behind the licencing choice, see Licenses.

7.1 Copyright Policy

John Hunter began matplotlib around 2003. Since shortly before his passing in 2012, Michael Droettboom has been the lead maintainer of matplotlib, but, as has always been the case, matplotlib is the work of many.

Prior to July of 2013, and the 1.3.0 release, the copyright of the source code was held by John Hunter. As of July 2013, and the 1.3.0 release, matplotlib has moved to a shared copyright model.

matplotlib uses a shared copyright model. Each contributor maintains copyright over their contributions to matplotlib. But, it is important to note that these contributions are typically only changes to the repositories. Thus, the matplotlib source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire matplotlib Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the matplotlib repositories.

The Matplotlib Development Team is the set of all contributors to the matplotlib project. A full list can be obtained from the git version control logs.

7.2 License agreement for matplotlib 1.4.3

1. This LICENSE AGREEMENT is between the Matplotlib Development Team (“MDT”), and the Individual or Organization (“Licensee”) accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, MDT hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 1.4.3 alone or in any derivative version, provided, however, that MDT’s License Agreement and MDT’s notice of copyright, i.e., “Copyright (c) 2012-2013
7.3 License agreement for matplotlib versions prior to 1.3.0

1. This LICENSE AGREEMENT is between John D. Hunter (“JDH”), and the Individual or Organization (“Licensee”) accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, JDH hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 1.4.3 alone or in any derivative version, provided, however, that JDH’s License Agreement and JDH’s notice of copyright, i.e., “Copyright (c) 2002-2009 John D. Hunter; All Rights Reserved” are retained in matplotlib 1.4.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 1.4.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 1.4.3.

4. JDH is making matplotlib 1.4.3 available to Licensee on an “AS IS” basis. JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 1.4.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 1.4.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 1.4.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between JDH and Licensee. This License Agreement does not grant permission to use JDH trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 1.4.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.
matplotlib was written by John Hunter and is now developed and maintained by a number of active developers. The current co-lead developers of matplotlib are Michael Droettboom and Thomas A. Caswell.

Special thanks to those who have made valuable contributions (roughly in order of first contribution by date). Any list like this is bound to be incomplete and can’t capture the thousands and thousands of contributions over the years from these and others:

Jeremy O’Donoghue wrote the wx backend

Andrew Straw provided much of the log scaling architecture, the fill command, PIL support for imshow, and provided many examples. He also wrote the support for dropped axis spines and the buildbot unit testing infrastructure which triggers the JPL/James Evans platform specific builds and regression test image comparisons from svn matplotlib across platforms on svn commits.

Charles Twardy provided the impetus code for the legend class and has made countless bug reports and suggestions for improvement.

Gary Ruben made many enhancements to errorbar to support x and y errorbar plots, and added a number of new marker types to plot.

John Gill wrote the table class and examples, helped with support for auto-legend placement, and added support for legending scatter plots.

David Moore wrote the paint backend (no longer used)

Todd Miller supported by STSCI contributed the TkAgg backend and the numerix module, which allows matplotlib to work with either numeric or numarray. He also ported image support to the postscript backend, with much pain and suffering.

Paul Barrett supported by STSCI overhauled font management to provide an improved, free-standing, platform independent font manager with a WC3 compliant font finder and cache mechanism and ported truetype and mathtext to PS.

Perry Greenfield supported by STSCI overhauled and modernized the goals and priorities page, implemented an improved colormap framework, and has provided many suggestions and a lot of insight to the overall design and organization of matplotlib.

Jared Wahlstrand wrote the initial SVG backend.

Steve Chaplin served as the GTK maintainer and wrote the Cairo and GTKCairo backends.

Jim Benson provided the patch to handle vertical mathtext.
Gregory Lielens provided the FltkAgg backend and several patches for the frontend, including contributions to toolbar2, and support for log ticking with alternate bases and major and minor log ticking.

Darren Dale did the work to do mathtext exponential labeling for log plots, added improved support for scalar formatting, and did the lions share of the psfrag LaTeX support for postscript. He has made substantial contributions to extending and maintaining the PS and Qt backends, and wrote the site.cfg and matplotlib.conf build and runtime configuration support. He setup the infrastructure for the sphinx documentation that powers the mpl docs.

Paul Mcguire provided the pyparsing module on which mathtext relies, and made a number of optimizations to the matplotlib mathtext grammar.

Fernando Perez has provided numerous bug reports and patches for cleaning up backend imports and expanding pylab functionality, and provided matplotlib support in the pylab mode for ipython. He also provided the matshow() command, and wrote TConfig, which is the basis for the experimental traited mpl configuration.

Andrew Dalke of Dalke Scientific Software contributed the strftime formatting code to handle years earlier than 1900.

Jochen Voss served as PS backend maintainer and has contributed several bugfixes.

Nadia Dencheva supported by STSCI provided the contouring and contour labeling code.

Baptiste Carvello provided the key ideas in a patch for proper shared axes support that underlies ganged plots and multiscale plots.

Jeffrey Whitaker at NOAA wrote the Basemap toolkit

Sigve Tjoraand, Ted Drain, James Evans and colleagues at the JPL collaborated on the QtAgg backend and sponsored development of a number of features including custom unit types, datetime support, scale free ellipses, broken bar plots and more. The JPL team wrote the unit testing image comparison infrastructure for regression test image comparisons.

James Amundson did the initial work porting the qt backend to qt4

Eric Firing has contributed significantly to contouring, masked array, pcolor, image and quiver support, in addition to ongoing support and enhancements in performance, design and code quality in most aspects of matplotlib.

Daishi Harada added support for “Dashed Text”. See dashpointlabel.py and TextWithDash.

Nicolas Young added support for byte images to imshow, which are more efficient in CPU and memory, and added support for irregularly sampled images.

The brainvisa Orsay team and Fernando Perez added Qt support to ipython in pylab mode.

Charlie Moad contributed work to matplotlib’s Cocoa support and has done a lot of work on the OSX and win32 binary releases.

Jouni K. Seppänen wrote the PDF backend and contributed numerous fixes to the code, to tex support and to the get_sample_data handler.
Paul Kienzle  improved the picking infrastructure for interactive plots, and with Alex Mont contributed fast rendering code for quadrilateral meshes.

Michael Droettboom supported by STSCI wrote the enhanced mathtext support, implementing Knuth’s box layout algorithms, saving to file-like objects across backends, and is responsible for numerous bug-fixes, much better font and unicode support, and feature and performance enhancements across the matplotlib code base. He also rewrote the transformation infrastructure to support custom projections and scales.

John Porter, Jonathon Taylor and Reinier Heeres  John Porter wrote the mplot3d module for basic 3D plotting in matplotlib, and Jonathon Taylor and Reinier Heeres ported it to the refactored transform trunk.

Jae-Joon Lee  Implemented fancy arrows and boxes, rewrote the legend support to handle multiple columns and fancy text boxes, wrote the axes grid toolkit, and has made numerous contributions to the code and documentation.

Paul Ivanov  Has worked on getting matplotlib integrated better with other tools, such as Sage and IPython, and getting the test infrastructure faster, lighter and meaner. Listen to his podcast.

Tony Yu  Has been involved in matplotlib since the early days, and recently has contributed stream plotting among many other improvements. He is the author of mpltools.

Michiel de Hoon  Wrote and maintains the macosx backend.

Ian Thomas  Contributed, among other things, the triangulation (tricolor and tripcontour) methods.

Benjamin Root  Has significantly improved the capabilities of the 3D plotting. He has improved matplotlib’s documentation and code quality throughout, and does invaluable triaging of pull requests and bugs.

Phil Elson  Fixed some deep-seated bugs in the transforms framework, and has been laser-focused on improving polish throughout matplotlib, tackling things that have been considered to large and daunting for a long time.

Damon McDougall  Added triangulated 3D surfaces and stack plots to matplotlib.
Part II

The Matplotlib FAQ
9.1 Report a compilation problem

See *Getting help*.

9.2 matplotlib compiled fine, but nothing shows up when I use it

The first thing to try is a *clean install* and see if that helps. If not, the best way to test your install is by running a script, rather than working interactively from a python shell or an integrated development
environment such as IDLE which add additional complexities. Open up a UNIX shell or a DOS command prompt and cd into a directory containing a minimal example in a file. Something like simple_plot.py for example:

```python
from pylab import *
plot([1,2,3])
show()
```

and run it with:

```bash
python simple_plot.py --verbose-helpful
```

This will give you additional information about which backends matplotlib is loading, version information, and more. At this point you might want to make sure you understand matplotlib’s configuration process, governed by the `matplotlibrc` configuration file which contains instructions within and the concept of the matplotlib backend.

If you are still having trouble, see *Getting help*.

### 9.3 How to completely remove matplotlib

Occasionally, problems with matplotlib can be solved with a clean installation of the package.

The process for removing an installation of matplotlib depends on how matplotlib was originally installed on your system. Follow the steps below that goes with your original installation method to cleanly remove matplotlib from your system.

#### 9.3.1 Easy Install

1. Delete the caches from your `.matplotlib configuration directory`.
2. Run:
   
   ```bash
easy_install -m matplotlib
   ```
3. Delete any `.egg` files or directories from your installation directory.

#### 9.3.2 Windows installer

1. Delete the caches from your `.matplotlib configuration directory`.
2. Use `Start → Control Panel` to start the Add and Remove Software utility.

#### 9.3.3 Source install

Unfortunately:
python setup.py clean
does not properly clean the build directory, and does nothing to the install directory. To cleanly rebuild:

1. Delete the caches from your `.matplotlib` configuration directory.
2. Delete the build directory in the source tree.
3. Delete any matplotlib directories or eggs from your installation directory.

### 9.4 How to Install

#### 9.4.1 Source install from git

Clone the main source using one of:

```
git clone git@github.com:matplotlib/matplotlib.git
```
or:

```
git clone git://github.com/matplotlib/matplotlib.git
```

and build and install as usual with:

```
> cd matplotlib
> python setup.py install
```

**Note:** If you are on Debian/Ubuntu, you can get all the dependencies required to build matplotlib with:

```
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing `yum-builddep` and then running:

```
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get all of the build dependencies, which will make building from source easier.

If you want to be able to follow the development branch as it changes just replace the last step with (make sure you have `setuptools` installed):

```
> python setup.py develop
```

This creates links in the right places and installs the command line script to the appropriate places.

**Note:** Mac OSX users please see the *Building on OSX* guide.

Windows users please see the *Building on Windows* guide.

Then, if you want to update your matplotlib at any time, just do:
> git pull

When you run `git pull`, if the output shows that only Python files have been updated, you are all set. If C files have changed, you need to run the `python setup.py develop` command again to compile them.

There is more information on using git in the developer docs.

### 9.5 Linux Notes

Because most Linux distributions use some sort of package manager, we do not provide a pre-built binary for the Linux platform. Instead, we recommend that you use the “Add Software” method for your system to install matplotlib. This will guarantee that everything that is needed for matplotlib will be installed as well.

If, for some reason, you can not use the package manager, Linux usually comes with at least a basic build system. Follow the instructions found above for how to build and install matplotlib.

### 9.6 OS-X Notes

#### 9.6.1 Which python for OS X?

Apple ships OS X with its own Python, in `/usr/bin/python`, and its own copy of matplotlib. Unfortunately, the way Apple currently installs its own copies of numpy, scipy and matplotlib means that these packages are difficult to upgrade (see system python packages). For that reason we strongly suggest that you install a fresh version of Python and use that as the basis for installing libraries such as numpy and matplotlib. One convenient way to install matplotlib with other useful Python software is to use one of the excellent Python scientific software collections that are now available:

- Anaconda from Continuum Analytics
- Canopy from Enthought

These collections include Python itself and a wide range of libraries; if you need a library that is not available from the collection, you can install it yourself using standard methods such as `pip`. Continuum and Enthought offer their own installation support for these collections; see the Anaconda and Canopy web pages for more information.

Other options for a fresh Python install are the standard installer from python.org, or installing Python using a general OSX package management system such as homebrew or macports. Power users on OSX will likely want one of homebrew or macports on their system to install open source software packages, but it is perfectly possible to use these systems with another source for your Python binary, such as Anaconda, Canopy or Python.org Python.

#### 9.6.2 Installing OSX binary wheels

If you are using recent Python from http://www.python.org, Macports or Homebrew, then you can use the standard pip installer to install matplotlib binaries in the form of wheels.
Python.org Python

Install pip following the standard pip install instructions. For the impatient, open a new Terminal.app window and:

curl -O https://bootstrap.pypa.io/get-pip.py

Then (Python 2.7):

python get-pip.py

or (Python 3):

python3 get-pip.py

You can now install matplotlib and all its dependencies with:

```
pip install matplotlib
```

Macports

For Python 2.7:

```
sudo port install py27-pip
sudo pip-2.7 install matplotlib
```

For Python 3.4:

```
sudo port install py34-pip
sudo pip-3.4 install matplotlib
```

Homebrew

For Python 2.7:

```
pip2 install matplotlib
```

For Python 3.4:

```
pip3 install matplotlib
```

You might also want to install IPython; we recommend you install IPython with the IPython notebook option, like this:

- Python.org Python: `pip install ipython[notebook]`
- Macports: `sudo pip-2.7 install ipython[notebook]` or `sudo pip-3.4 install ipython[notebook]`
- Homebrew: `pip2 install ipython[notebook]` or `pip3 install ipython[notebook]`
Pip problems

If you get errors with pip trying to run a compiler like gcc or clang, then the first thing to try is to install xcode and retry the install. If that does not work, then check Getting help.

9.6.3 Installing via OSX mpkg installer package

matplotlib also has a disk image (.dmg) installer, which contains a typical Installer.app package to install matplotlib. You should use binary wheels instead of the disk image installer if you can, because:

- wheels work with Python.org Python, homebrew and macports, the disk image installer only works with Python.org Python.
- The disk image installer doesn’t check for recent versions of packages that matplotlib depends on, and unconditionally installs the versions of dependencies contained in the disk image installer. This can overwrite packages that you have already installed, which might cause problems for other packages, if you have a pre-existing Python.org setup on your computer.

If you still want to use the disk image installer, read on.

**Note:** Before installing via the disk image installer, be sure that all of the packages were compiled for the same version of python. Often, the download site for NumPy and matplotlib will display a supposed ‘current’ version of the package, but you may need to choose a different package from the full list that was built for your combination of python and OSX.

The disk image installer will have a .dmg extension, and will have a name like matplotlib-1.4.0-py2.7-macosx10.6.dmg. The name of the installer depends on the versions of python and matplotlib it was built for, and the version of OSX that the matching Python.org installer was built for. For example, if the matching Python.org Python installer was built for OSX 10.6 or greater, the dmg file will end in -macosx10.6.dmg. You need to download this disk image file, open the disk image file by double clicking, and find the new matplotlib disk image icon on your desktop. Double click on that icon to show the contents of the image. Then double-click on the .mpkg icon, which will have a name like matplotlib-1.4.0-py2.7-macosx10.6.mpkg, it will run the Installer.app, prompt you for a password if you need system-wide installation privileges, and install to a directory like /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages
(exact path depends on your Python version).

9.6.4 Checking your installation

The new version of matplotlib should now be on your Python “path”. Check this with one of these commands at the Terminal.app command line:

```python
python2.7 -c 'import matplotlib; print(matplotlib.__version__, matplotlib.__file__)'
```

(Python 2.7) or:

```python
python3.4 -c 'import matplotlib; print(matplotlib.__version__, matplotlib.__file__)'
```

(Python 3.4). You should see something like this:
1.4.0 /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/matplotlib/__init__.pyc

where 1.4.0 is the matplotlib version you just installed, and the path following depends on whether you are using Python.org Python, Homebrew or Macports. If you see another version, or you get an error like this:

```
Traceback (most recent call last):
  File "<string>" , line 1, in <module>
ImportError: No module named matplotlib
```

then check that the Python binary is the one you expected by doing one of these commands in Terminal.app:

```
which python2.7
```

or:

```
which python3.4
```

If you get the result `/usr/bin/python2.7`, then you are getting the Python installed with OSX, which is probably not what you want. Try closing and restarting Terminal.app before running the check again. If that doesn’t fix the problem, depending on which Python you wanted to use, consider reinstalling Python.org Python, or check your homebrew or macports setup. Remember that the disk image installer only works for Python.org Python, and will not get picked up by other Pythons. If all these fail, please let us know: see Getting help.

### 9.7 Windows Notes

We recommend you use one of the excellent python collections which include Python itself and a wide range of libraries including matplotlib:

- **Anaconda** from Continuum Analytics
- **Canopy** from Enthought
- **Python (x, y)**

Python (X, Y) is Windows-only, whereas Anaconda and Canopy are cross-platform.

#### 9.7.1 Standalone binary installers for Windows

If you have already installed Python and numpy, you can use one of the matplotlib binary installers for windows – you can get these from the [download site](https://matplotlib.org). Chose the files with an `.exe` extension that match your version of Python (e.g., `py2.7` if you installed Python 2.7). If you haven’t already installed Python, you can get the official version from the [Python web site](https://www.python.org).
10.1 General Concepts

matplotlib has an extensive codebase that can be daunting to many new users. However, most of matplotlib can be understood with a fairly simple conceptual framework and knowledge of a few important points.

Plotting requires action on a range of levels, from the most general (e.g., ‘contour this 2-D array’) to the most specific (e.g., ‘color this screen pixel red’). The purpose of a plotting package is to assist you in visualizing your data as easily as possible, with all the necessary control – that is, by using relatively high-level commands most of the time, and still have the ability to use the low-level commands when needed.

Therefore, everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib “state-machine environment” which is provided by the matplotlib.pyplot module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure.
Note: Pyplot’s state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.

For even more control – which is essential for things like embedding matplotlib plots in GUI applications – the pyplot level may be dropped completely, leaving a purely object-oriented approach.

10.2 Parts of a Figure
10.2.1 Figure

The **whole** figure (marked as the outer red box). The figure keeps track of all the child **Axes**, a smattering of ‘special’ artists (titles, figure legends, etc), and the **canvas**. (Don’t worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can have any number of **Axes**, but to be useful should have at least one.

The easiest way to create a new figure is with pyplot:

```python
fig = plt.figure() # an empty figure with no axes
fig, ax_lst = plt.subplots(2, 2) # a figure with a 2x2 grid of Axes
```

10.2.2 Axes

This is what you think of as ‘a plot’, it is the region of the image with the data space (marked as the inner blue box). A given figure can contain many Axes, but a given **Axes** object can only be in one **Figure**. The Axes contains two (or three in the case of 3D) **Axis** objects (be aware of the difference between **Axes** and **Axis**) which take care of the data limits (the data limits can also be controlled via set via the `set_xlim()` and `set_ylim()` Axes methods). Each **Axes** has a title (set via `set_title()`), an x-label (set via `set_xlabel()`), and a y-label set via `set_ylabel()`.

The **Axes** class and it’s member functions are the primary entry point to working with the OO interface.

10.2.3 Axis

These are the number-line-like objects (circled in green). They take care of setting the graph limits and generating the ticks (the marks on the axis) and ticklabels (strings labeling the ticks). The location of the ticks is determined by a **Locator** object and the ticklabel strings are formatted by a **Formatter**. The combination of the correct **Locator** and **Formatter** gives very fine control over the tick locations and labels.

10.2.4 Artist

Basically everything you can see on the figure is an artist (even the **Figure**, **Axes**, and **Axis** objects). This includes **Text** objects, **Line2D** objects, **collection** objects, **Patch** objects ... (you get the idea). When the figure is rendered, all of the artists are drawn to the **canvas**. Most Artists are tied to an **Axes**; such an Artist cannot be shared by multiple **Axes**, or moved from one to another.

10.3 Types of inputs to plotting functions

All of plotting functions expect `np.array` or `np.ma.masked_array` as input. Classes that are ‘array-like’ such as pandas data objects and `np.matrix` may or may not work as intended. It is best to convert these to `np.array` objects prior to plotting.

For example, to covert a `pandas.DataFrame`
a = pandas.DataFrame(np.random.rand(4,5), columns = list('abcde'))
a_asndarray = a.values

and to covert a np.matrix
b = np.matrix([[1,2],[3,4]])
b_asarray = np.asarray(b)

10.4 Matplotlib, pyplot and pylab: how are they related?

Matplotlib is the whole package; matplotlib.pyplot is a module in matplotlib; and pylab is a module that gets installed alongside matplotlib.

Pyplot provides the state-machine interface to the underlying object-oriented plotting library. The state-machine implicitly and automatically creates figures and axes to achieve the desired plot. For example:

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
plt.show()
```

The first call to `plt.plot` will automatically create the necessary figure and axes to achieve the desired plot. Subsequent calls to `plt.plot` re-use the current axes and each add another line. Setting the title, legend, and axis labels also automatically use the current axes and set the title, create the legend, and label the axis respectively.

`pylab` is a convenience module that bulk imports `matplotlib.pyplot` (for plotting) and `numpy` (for mathematics and working with arrays) in a single name space. Although many examples use `pylab`, it is no longer recommended.

For non-interactive plotting it is suggested to use `pyplot` to create the figures and then the OO interface for plotting.
10.5 Coding Styles

When viewing this documentation and examples, you will find different coding styles and usage patterns. These styles are perfectly valid and have their pros and cons. Just about all of the examples can be converted into another style and achieve the same results. The only caveat is to avoid mixing the coding styles for your own code.

**Note:** Developers for matplotlib have to follow a specific style and guidelines. See *The Matplotlib Developers' Guide*.

Of the different styles, there are two that are officially supported. Therefore, these are the preferred ways to use matplotlib.

For the pyplot style, the imports at the top of your scripts will typically be:

```python
import matplotlib.pyplot as plt
import numpy as np
```

Then one calls, for example, `np.arange`, `np.zeros`, `np.pi`, `plt.figure`, `plt.plot`, `plt.show`, etc. Use the pyplot interface for creating figures, and then use the object methods for the rest:

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.2)
y = np.sin(x)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
plt.show()
```

So, why all the extra typing instead of the MATLAB-style (which relies on global state and a flat namespace)? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, more clear as to where things come from and what is going on. For more complicated applications, this explicitness and clarity becomes increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and maintain.

Typically one finds oneself making the same plots over and over again, but with different data sets, which leads to needing to write specialized functions to do the plotting. The recommended function signature is something like:

```python
def my_plotter(ax, data1, data2, param_dict):
    
    A helper function to make a graph

    Parameters
    ----------
    ax : Axes
        The axes to draw to

    data1 : array
        The x data
```

```python
    
    data2 : array
        The y data
```
data2 : array
    The y data

param_dict : dict
    Dictionary of kwargs to pass to ax.plot

Returns
-------
out : list
    list of artists added

out = ax.plot(data1, data2, **param_dict)
return out

which you would then use as:

fig, ax = plt.subplots(1, 1)
my_plotter(ax, data1, data2, {'marker':'x'})

or if you wanted to have 2 sub-plots:

fig, (ax1, ax2) = plt.subplots(1, 2)
my_plotter(ax1, data1, data2, {'marker':'x'})
my_plotter(ax2, data3, data4, {'marker':'o'})

Again, for these simple examples this style seems like overkill, however once the graphs get slightly more complex it pays off.

10.6 What is a backend?

A lot of documentation on the website and in the mailing lists refers to the “backend” and many new users are confused by this term. matplotlib targets many different use cases and output formats. Some people use matplotlib interactively from the python shell and have plotting windows pop up when they type commands. Some people embed matplotlib into graphical user interfaces like wxpython or pygtk to build rich applications. Others use matplotlib in batch scripts to generate postscript images from some numerical simulations, and still others in web application servers to dynamically serve up graphs.

To support all of these use cases, matplotlib can target different outputs, and each of these capabilities is called a backend; the “frontend” is the user facing code, i.e., the plotting code, whereas the “backend” does all the hard work behind-the-scenes to make the figure. There are two types of backends: user interface backends (for use in pygtk, wxpython, tkinter, qt4, or macosx; also referred to as “interactive backends”) and hardcopy backends to make image files (PNG, SVG, PDF, PS; also referred to as “non-interactive backends”).

There are a two primary ways to configure your backend. One is to set the backend parameter in your matplotlibrc file (see Customizing matplotlib):

backend : WXAgg  # use wxpython with antigrain (agg) rendering

The other is to use the matplotlib use() directive:
import matplotlib
matplotlib.use('PS')  # generate postscript output by default

If you use the use directive, this must be done before importing matplotlib.pyplot or matplotlib.pylab.

Note: Backend name specifications are not case-sensitive; e.g., ‘GTKAgg’ and ‘gtkagg’ are equivalent.

With a typical installation of matplotlib, such as from a binary installer or a linux distribution package, a good default backend will already be set, allowing both interactive work and plotting from scripts, with output to the screen and/or to a file, so at least initially you will not need to use either of the two methods given above.

If, however, you want to write graphical user interfaces, or a web application server (Matplotlib in a web application server), or need a better understanding of what is going on, read on. To make things a little more customizable for graphical user interfaces, matplotlib separates the concept of the renderer (the thing that actually does the drawing) from the canvas (the place where the drawing goes). The canonical renderer for user interfaces is Agg which uses the Anti-Grain Geometry C++ library to make a raster (pixel) image of the figure. All of the user interfaces except macosx can be used with agg rendering, e.g., WXAgg, GTKAgg, QT4Agg, TkAgg. In addition, some of the user interfaces support other rendering engines. For example, with GTK, you can also select GDK rendering (backend GTK) or Cairo rendering (backend GTKCairo).

For the rendering engines, one can also distinguish between vector or raster renderers. Vector graphics languages issue drawing commands like “draw a line from this point to this point” and hence are scale free, and raster backends generate a pixel representation of the line whose accuracy depends on a DPI setting.

Here is a summary of the matplotlib renderers (there is an eponymous backed for each; these are non-interactive backends, capable of writing to a file):

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Filetypes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGG</td>
<td>png</td>
<td>raster graphics – high quality images using the Anti-Grain Geometry engine</td>
</tr>
<tr>
<td>PS</td>
<td>ps eps</td>
<td>vector graphics – Postscript output</td>
</tr>
<tr>
<td>PDF</td>
<td>pdf</td>
<td>vector graphics – Portable Document Format</td>
</tr>
<tr>
<td>SVG</td>
<td>svg</td>
<td>vector graphics – Scalable Vector Graphics</td>
</tr>
<tr>
<td>Cairo</td>
<td>png ps pdf svg</td>
<td>vector graphics – Cairo graphics</td>
</tr>
<tr>
<td>GDK</td>
<td>png jpg tiff</td>
<td>raster graphics – the Gimp Drawing Kit</td>
</tr>
</tbody>
</table>

And here are the user interfaces and renderer combinations supported; these are interactive backends, capable of displaying to the screen and of using appropriate renderers from the table above to write to a file:

10.6. What is a backend?
Matplotlib, Release 1.4.3

<table>
<thead>
<tr>
<th>Backend</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTK-</td>
<td>Agg rendering to a GTK 2.x canvas (requires PyGTK)</td>
</tr>
<tr>
<td>Agg</td>
<td></td>
</tr>
<tr>
<td>GTK3A</td>
<td>Agg rendering to a GTK 3.x canvas (requires PyGObject)</td>
</tr>
<tr>
<td>GTK</td>
<td>GDK rendering to a GTK 2.x canvas (not recommended) (requires PyGTK)</td>
</tr>
<tr>
<td>GTK-C</td>
<td>Cairo rendering to a GTK 2.x canvas (requires PyGTK and pycairo)</td>
</tr>
<tr>
<td>GTK3C</td>
<td>Cairo rendering to a GTK 3.x canvas (requires PyGObject and pycairo)</td>
</tr>
<tr>
<td>Cairo</td>
<td></td>
</tr>
<tr>
<td>WXAgg</td>
<td>Agg rendering to a wxWidgets canvas (requires wxPython)</td>
</tr>
<tr>
<td>WX</td>
<td>Native wxWidgets drawing to a wxWidgets Canvas (not recommended) (requires</td>
</tr>
<tr>
<td>TkAgg</td>
<td>Tk canvas (requires TkInter)</td>
</tr>
<tr>
<td>Qt4Agg</td>
<td>Agg rendering to a Qt4 canvas (requires PyQt4)</td>
</tr>
<tr>
<td>macOSx</td>
<td>Cocoa rendering in OSX windows (presently lacks blocking show() behavior when</td>
</tr>
<tr>
<td></td>
<td>matplotlib is in non-interactive mode)</td>
</tr>
</tbody>
</table>

10.7 How do I select PyQt4 or PySide?

You can choose either PyQt4 or PySide when using the qt4 backend by setting the appropriate value for backend.qt4 in your matplotlibrc file. The default value is PyQt4.

The setting in your matplotlibrc file can be overridden by setting the QT_API environment variable to either pyqt or pyside to use PyQt4 or PySide, respectively.

Since the default value for the bindings to be used is PyQt4, matplotlib first tries to import it, if the import fails, it tries to import PySide.

10.8 What is interactive mode?

Use of an interactive backend (see What is a backend?) permits—but does not by itself require or ensure—plotting to the screen. Whether and when plotting to the screen occurs, and whether a script or shell session continues after a plot is drawn on the screen, depends on the functions and methods that are called, and on a state variable that determines whether matplotlib is in “interactive mode”. The default Boolean value is set by the matplotlibrc file, and may be customized like any other configuration parameter (see Customizing matplotlib). It may also be set via matplotlib.interactive(), and its value may be queried via matplotlib.is_interactive(). Turning interactive mode on and off in the middle of a stream of plotting commands, whether in a script or in a shell, is rarely needed and potentially confusing, so in the following we will assume all plotting is done with interactive mode either on or off.

**Note:** Major changes related to interactivity, and in particular the role and behavior of show(), were made in the transition to matplotlib version 1.0, and bugs were fixed in 1.0.1. Here we describe the version 1.0.1 behavior for the primary interactive backends, with the partial exception of macOSx.

Interactive mode may also be turned on via matplotlib.pyplot.ion(), and turned off via matplotlib.pyplot.ioff().

**Note:** Interactive mode works with suitable backends in ipython and in the ordinary python shell, but it
does *not* work in the IDLE IDE.

## 10.8.1 Interactive example

From an ordinary python prompt, or after invoking ipython with no options, try this:

```python
import matplotlib.pyplot as plt
plt.ion()
plt.plot([1.6, 2.7])
```

Assuming you are running version 1.0.1 or higher, and you have an interactive backend installed and selected by default, you should see a plot, and your terminal prompt should also be active; you can type additional commands such as:

```python
plt.title("interactive test")
plt.xlabel("index")
```

and you will see the plot being updated after each line. This is because you are in interactive mode *and* you are using pyplot functions. Now try an alternative method of modifying the plot. Get a reference to the Axes instance, and call a method of that instance:

```python
ax = plt.gca()
ax.plot([3.1, 2.2])
```

Nothing changed, because the Axes methods do not include an automatic call to `draw_if_interactive()`; that call is added by the pyplot functions. If you are using methods, then when you want to update the plot on the screen, you need to call `draw()`:

```python
plt.draw()
```

Now you should see the new line added to the plot.

## 10.8.2 Non-interactive example

Start a fresh session as in the previous example, but now turn interactive mode off:

```python
import matplotlib.pyplot as plt
plt.ioff()
plt.plot([1.6, 2.7])
```

Nothing happened—or at least nothing has shown up on the screen (unless you are using `macosx` backend, which is anomalous). To make the plot appear, you need to do this:

```python
plt.show()
```

Now you see the plot, but your terminal command line is unresponsive; the `show()` command *blocks* the input of additional commands until you manually kill the plot window.

What good is this—being forced to use a blocking function? Suppose you need a script that plots the contents of a file to the screen. You want to look at that plot, and then end the script. Without some blocking
command such as show(), the script would flash up the plot and then end immediately, leaving nothing on the screen.

In addition, non-interactive mode delays all drawing until show() is called; this is more efficient than re-drawing the plot each time a line in the script adds a new feature.

Prior to version 1.0, show() generally could not be called more than once in a single script (although sometimes one could get away with it); for version 1.0.1 and above, this restriction is lifted, so one can write a script like this:

```python
import numpy as np
import matplotlib.pyplot as plt
plt.ioff()
for i in range(3):
    plt.plot(np.random.rand(10))
plt.show()
```

which makes three plots, one at a time.

10.8.3 Summary

In interactive mode, pyplot functions automatically draw to the screen.

When plotting interactively, if using object method calls in addition to pyplot functions, then call `draw()` whenever you want to refresh the plot.

Use non-interactive mode in scripts in which you want to generate one or more figures and display them before ending or generating a new set of figures. In that case, use `show()` to display the figure(s) and to block execution until you have manually destroyed them.
Contents

- How-To
  - Plotting: howto
    * Find all objects in a figure of a certain type
    * How to prevent ticklabels from having an offset
    * Save transparent figures
    * Save multiple plots to one pdf file
    * Move the edge of an axes to make room for tick labels
    * Automatically make room for tick labels
    * Configure the tick linewidths
    * Align my ylabels across multiple subplots
    * Skip dates where there is no data
    * Control the depth of plot elements
    * Make the aspect ratio for plots equal
    * Multiple y-axis scales
    * Generate images without having a window appear
    * Use show()
  - Contributing: howto
    * Request a new feature
    * Reporting a bug or submitting a patch
    * Contribute to matplotlib documentation
  - Matplotlib in a web application server
    * matplotlib with apache
    * matplotlib with django
    * matplotlib with zope
    * Clickable images for HTML
  - Search examples
  - Cite Matplotlib
11.1 Plotting: howto

11.1.1 Find all objects in a figure of a certain type

Every matplotlib artist (see Artist tutorial) has a method called findobj() that can be used to recursively search the artist for any artists it may contain that meet some criteria (e.g., match all Line2D instances or match some arbitrary filter function). For example, the following snippet finds every object in the figure which has a set_color property and makes the object blue:

```python
def myfunc(x):
    return hasattr(x, 'set_color')

for o in fig.findobj(myfunc):
    o.set_color('blue')
```

You can also filter on class instances:

```python
import matplotlib.text as text
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

11.1.2 How to prevent ticklabels from having an offset

The default formatter will use an offset to reduce the length of the ticklabels. To turn this feature off on a per-axis basis:

```python
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```

set the rcParam axes.formatter.useoffset, or use a different formatter. See ticker for details.

11.1.3 Save transparent figures

The savefig() command has a keyword argument transparent which, if ‘True’, will make the figure and axes backgrounds transparent when saving, but will not affect the displayed image on the screen.

If you need finer grained control, e.g., you do not want full transparency or you want to affect the screen displayed version as well, you can set the alpha properties directly. The figure has a Rectangle instance called patch and the axes has a Rectangle instance called patch. You can set any property on them directly (facecolor, edgecolor, linewidth, linestyle, alpha). e.g.:

```python
fig = plt.figure()
fig.patch.set_alpha(0.5)
ax = fig.add_subplot(111)
ax.patch.set_alpha(0.5)
```

If you need all the figure elements to be transparent, there is currently no global alpha setting, but you can set the alpha channel on individual elements, e.g.:}
ax.plot(x, y, alpha=0.5)
ax.set_xlabel('volts', alpha=0.5)

11.1.4 Save multiple plots to one pdf file

Many image file formats can only have one image per file, but some formats support multi-page files. Currently only the pdf backend has support for this. To make a multi-page pdf file, first initialize the file:

```python
from matplotlib.backends.backend_pdf import PdfPages
pp = PdfPages('multipage.pdf')
```

You can give the `PdfPages` object to `savefig()`, but you have to specify the format:

```python
plt.savefig(pp, format='pdf')
```

An easier way is to call `PdfPages.savefig`:

```python
pp.savefig()
```

Finally, the multipage pdf object has to be closed:

```python
pp.close()
```

11.1.5 Move the edge of an axes to make room for tick labels

For subplots, you can control the default spacing on the left, right, bottom, and top as well as the horizontal and vertical spacing between multiple rows and columns using the `matplotlib.figure.Figure.subplots_adjust()` method (in pyplot it is `subplots_adjust()`). For example, to move the bottom of the subplots up to make room for some rotated x tick labels:

```python
fig = plt.figure()
fig.subplots_adjust(bottom=0.2)
ax = fig.add_subplot(111)
```

You can control the defaults for these parameters in your `matplotlibrc` file; see `Customizing matplotlib`. For example, to make the above setting permanent, you would set:

```python
figure.subplot.bottom : 0.2  # the bottom of the subplots of the figure
```

The other parameters you can configure are, with their defaults

- `left` = 0.125  the left side of the subplots of the figure
- `right` = 0.9  the right side of the subplots of the figure
- `bottom` = 0.1  the bottom of the subplots of the figure
- `top` = 0.9  the top of the subplots of the figure
- `wspace` = 0.2  the amount of width reserved for blank space between subplots
- `hspace` = 0.2  the amount of height reserved for white space between subplots
If you want additional control, you can create an Axes using the axes() command (or equivalently the figure add_axes() method), which allows you to specify the location explicitly:

```python
ax = fig.add_axes([left, bottom, width, height])
```

where all values are in fractional (0 to 1) coordinates. See `pylab_examples` example code: `axes_demo.py` for an example of placing axes manually.

### 11.1.6 Automatically make room for tick labels

**Note:** This is now easier to handle than ever before. Calling `tight_layout()` can fix many common layout issues. See the [Tight Layout guide](#).

The information below is kept here in case it is useful for other purposes.

In most use cases, it is enough to simply change the subplots adjust parameters as described in *Move the edge of an axes to make room for tick labels*. But in some cases, you don’t know ahead of time what your tick labels will be, or how large they will be (data and labels outside your control may be being fed into your graphing application), and you may need to automatically adjust your subplot parameters based on the size of the tick labels. Any Text instance can report its extent in window coordinates (a negative x coordinate is outside the window), but there is a rub.

The [RendererBase](#) instance, which is used to calculate the text size, is not known until the figure is drawn (`draw()`). After the window is drawn and the text instance knows its renderer, you can call `get_window_extent()`. One way to solve this chicken and egg problem is to wait until the figure is drawn by connecting (mpl_connect()) to the “on_draw” signal ([DrawEvent](#)) and get the window extent there, and then do something with it, e.g., move the left of the canvas over; see [Event handling and picking](#).

Here is an example that gets a bounding box in relative figure coordinates (0..1) of each of the labels and uses it to move the left of the subplots over so that the tick labels fit in the figure:

```python
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))
ax.set_yticks((2, 5, 7))
labels = ax.set_yticklabels(('really, really, really', 'long', 'labels'))

def on_draw(event):
    bboxes = []
    for label in labels:
        bbox = label.get_window_extent()
        # the figure transform goes from relative coords->pixels and we
        # want the inverse of that
        bboxi = bbox.inverse_transformed(fig.transFigure)
        bboxes.append(bboxi)

        # this is the bbox that bounds all the bboxes, again in relative
        # figure coords
        bbox = mtransforms.Bbox.union(bboxes)
```

---

368 Chapter 11. How-To
```python
if fig.subplotpars.left < bbox.width:
    # we need to move it over
    fig.subplots_adjust(left=1.1*bbox.width) # pad a little
    fig.canvas.draw()

fig.canvas.mpl_connect('draw_event', on_draw)

plt.show()
```

### 11.1.7 Configure the tick linewidths

In matplotlib, the ticks are *markers*. All `Line2D` objects support a line (solid, dashed, etc) and a marker (circle, square, tick). The tick linewidth is controlled by the “markeredgewidth” property:

```python
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))

for line in ax.get_xticklines() + ax.get_yticklines():
    line.set_markersize(10)

plt.show()
```
The other properties that control the tick marker, and all markers, are `markerfacecolor`, `markeredgecolor`, `markeredgewidth`, `markersize`. For more information on configuring ticks, see `Axis containers` and `Tick containers`.

### 11.1.8 Align my ylabels across multiple subplots

If you have multiple subplots over one another, and the y data have different scales, you can often get ylabels that do not align vertically across the multiple subplots, which can be unattractive. By default, matplotlib positions the x location of the ylabel so that it does not overlap any of the y ticks. You can override this default behavior by specifying the coordinates of the label. The example below shows the default behavior in the left subplots, and the manual setting in the right subplots.

```python
import numpy as np
import matplotlib.pyplot as plt

box = dict(facecolor='yellow', pad=5, alpha=0.2)

fig = plt.figure()
fig.subplots_adjust(left=0.2, wspace=0.6)

ax1 = fig.add_subplot(221)
ax1.plot(2000*np.random.rand(10))
ax1.set_title('ylabels not aligned')
ax1.set_ylabel('misaligned 1', bbox=box)
ax1.set_ylim(0, 2000)

ax2 = fig.add_subplot(222)
ax2.set_title('ylabels aligned')
ax2.plot(2000*np.random.rand(10))
ax2.set_ylabel('aligned 1', bbox=box)
ax2.yaxis.set_label_coords(-0.3, 0.5)
ax2.set_ylim(0, 2000)

ax3 = fig.add_subplot(223)
ax3.plot(np.random.rand(10))
ax3.set_ylabel('misaligned 2', bbox=box)

ax4 = fig.add_subplot(224)
ax4.plot(np.random.rand(10))
ax4.set_ylabel('aligned 2', bbox=box)
ax4.yaxis.set_label_coords(-0.3, 0.5)

plt.show()
```
11.1.9 Skip dates where there is no data

When plotting time series, e.g., financial time series, one often wants to leave out days on which there is no data, e.g., weekends. By passing in dates on the x-axis, you get large horizontal gaps on periods when there is not data. The solution is to pass in some proxy x-data, e.g., evenly sampled indices, and then use a custom formatter to format these as dates. The example below shows how to use an ‘index formatter’ to achieve the desired plot:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.ticker as ticker

r = mlab.csv2rec('../data/aapl.csv')
r.sort()
r = r[-30:]  # get the last 30 days
N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices

def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y-%m-%d')
```

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.ticker as ticker

r = mlab.csv2rec('../data/aapl.csv')
r.sort()
r = r[-30:]  # get the last 30 days
N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices

def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y-%m-%d')
```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(ind, r.adj_close, 'o-')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
fig.autofmt_xdate()
plt.show()

11.1.10 Control the depth of plot elements

Within an axes, the order that the various lines, markers, text, collections, etc appear is determined by the `set_zorder()` property. The default order is patches, lines, text, with collections of lines and collections of patches appearing at the same level as regular lines and patches, respectively:

```python
line, = ax.plot(x, y, zorder=10)
```

You can also use the Axes property `set_axisbelow()` to control whether the grid lines are placed above or below your other plot elements.

11.1.11 Make the aspect ratio for plots equal

The Axes property `set_aspect()` controls the aspect ratio of the axes. You can set it to be ‘auto’, ‘equal’, or some ratio which controls the ratio:

```python
ax = fig.add_subplot(111, aspect='equal')
```

11.1.12 Multiple y-axis scales

A frequent request is to have two scales for the left and right y-axis, which is possible using `twinx()` (more than two scales are not currently supported, though it is on the wish list). This works pretty well, though there are some quirks when you are trying to interactively pan and zoom, because both scales do not get the signals.

The approach uses `twinx()` (and its sister `twiny()`) to use 2 different axes, turning the axes rectangular frame off on the 2nd axes to keep it from obscuring the first, and manually setting the tick locs and labels as desired. You can use separate matplotlib.ticker formatters and locators as desired because the two axes are independent.
11.1.13 Generate images without having a window appear

The easiest way to do this is use a non-interactive backend (see What is a backend?) such as Agg (for PNGs), PDF, SVG or PS. In your figure-generating script, just call the `matplotlib.use()` directive before importing pylab or pyplot:

```python
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.plot([1,2,3])
plt.savefig('myfig')
```

See also:

*Matplotlib in a web application server* for information about running matplotlib inside of a web application.

11.1.14 Use `show()`

When you want to view your plots on your display, the user interface backend will need to start the GUI mainloop. This is what `show()` does. It tells matplotlib to raise all of the figure windows created so far and start the mainloop. Because this mainloop is blocking by default (i.e., script execution is paused), you should only call this once per script, at the end. Script execution is resumed after the last window is closed.
Therefore, if you are using matplotlib to generate only images and do not want a user interface window, you do not need to call `show` (see `Generate images without having a window appear` and `What is a backend?`).

**Note:** Because closing a figure window invokes the destruction of its plotting elements, you should call `savefig()` before calling `show` if you wish to save the figure as well as view it.

New in version v1.0.0: `show` now starts the GUI mainloop only if it isn’t already running. Therefore, multiple calls to `show` are now allowed.

Having `show` block further execution of the script or the python interpreter depends on whether matplotlib is set for interactive mode or not. In non-interactive mode (the default setting), execution is paused until the last figure window is closed. In interactive mode, the execution is not paused, which allows you to create additional figures (but the script won’t finish until the last figure window is closed).

**Note:** Support for interactive/non-interactive mode depends upon the backend. Until version 1.0.0 (and subsequent fixes for 1.0.1), the behavior of the interactive mode was not consistent across backends. As of v1.0.1, only the macosx backend differs from other backends because it does not support non-interactive mode.

Because it is expensive to draw, you typically will not want matplotlib to redraw a figure many times in a script such as the following:

```python
plot([1,2,3])  # draw here ?
xlabel('time')    # and here ?
ylabel('volts')  # and here ?
title('a simple plot') # and here ?
show()
```

However, it is possible to force matplotlib to draw after every command, which might be what you want when working interactively at the python console (see `Using matplotlib in a python shell`), but in a script you want to defer all drawing until the call to `show`. This is especially important for complex figures that take some time to draw. `show()` is designed to tell matplotlib that you’re all done issuing commands and you want to draw the figure now.

**Note:** `show()` should typically only be called at most once per script and it should be the last line of your script. At that point, the GUI takes control of the interpreter. If you want to force a figure draw, use `draw()` instead.

Many users are frustrated by `show` because they want it to be a blocking call that raises the figure, pauses the script until they close the figure, and then allow the script to continue running until the next figure is created and the next show is made. Something like this:

```python
# WARNING : illustrating how NOT to use show
for i in range(10):
    # make figure i
    show()
```

This is not what `show` does and unfortunately, because doing blocking calls across user interfaces can be tricky, is currently unsupported, though we have made significant progress towards supporting blocking events.
New in version v1.0.0: As noted earlier, this restriction has been relaxed to allow multiple calls to `show`. In most backends, you can now expect to be able to create new figures and raise them in a subsequent call to `show` after closing the figures from a previous call to `show`.

11.2 Contributing: howto

11.2.1 Request a new feature

Is there a feature you wish matplotlib had? Then ask! The best way to get started is to email the developer mailing list for discussion. This is an open source project developed primarily in the contributors free time, so there is no guarantee that your feature will be added. The best way to get the feature you need added is to contribute it yourself.

11.2.2 Reporting a bug or submitting a patch

The development of matplotlib is organized through github. If you would like to report a bug or submit a patch please use that interface.

To report a bug create an issue on github (this requires having a github account). Please include a Short, Self Contained, Correct (Compilable), Example demonstrating what the bug is. Including a clear, easy to test example makes it easy for the developers to evaluate the bug. Expect that the bug reports will be a conversation. If you do not want to register with github, please email bug reports to the mailing list.

The easiest way to submit patches to matplotlib is through pull requests on github. Please see the The Matplotlib Developers’ Guide for the details.

11.2.3 Contribute to matplotlib documentation

matplotlib is a big library, which is used in many ways, and the documentation has only scratched the surface of everything it can do. So far, the place most people have learned all these features are through studying the examples (Search examples), which is a recommended and great way to learn, but it would be nice to have more official narrative documentation guiding people through all the dark corners. This is where you come in.

There is a good chance you know more about matplotlib usage in some areas, the stuff you do every day, than many of the core developers who wrote most of the documentation. Just pulled your hair out compiling matplotlib for windows? Write a FAQ or a section for the Installation page. Are you a digital signal processing wizard? Write a tutorial on the signal analysis plotting functions like `xcorr()`, `psd()` and `specgram()`. Do you use matplotlib with django or other popular web application servers? Write a FAQ or tutorial and we’ll find a place for it in the User’s Guide. Bundle matplotlib in a py2exe app? ... I think you get the idea.

matplotlib is documented using the sphinx extensions to restructured text (ReST). sphinx is an extensible python framework for documentation projects which generates HTML and PDF, and is pretty easy to write; you can see the source for this document or any page on this site by clicking on the Show Source link at the end of the page in the sidebar (or here for this document).
The sphinx website is a good resource for learning sphinx, but we have put together a cheat-sheet at Documenting matplotlib which shows you how to get started, and outlines the matplotlib conventions and extensions, e.g., for including plots directly from external code in your documents.

Once your documentation contributions are working (and hopefully tested by actually building the docs) you can submit them as a patch against git. See Install git and Reporting a bug or submitting a patch. Looking for something to do? Search for TODO or look at the open issues on github.

11.3 Matplotlib in a web application server

Many users report initial problems trying to use matplotlib in web application servers, because by default matplotlib ships configured to work with a graphical user interface which may require an X11 connection. Since many barebones application servers do not have X11 enabled, you may get errors if you don’t configure matplotlib for use in these environments. Most importantly, you need to decide what kinds of images you want to generate (PNG, PDF, SVG) and configure the appropriate default backend. For 99% of users, this will be the Agg backend, which uses the C++ antigrain rendering engine to make nice PNGs. The Agg backend is also configured to recognize requests to generate other output formats (PDF, PS, EPS, SVG). The easiest way to configure matplotlib to use Agg is to call:

```
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

For more on configuring your backend, see What is a backend?.

Alternatively, you can avoid pylab/pyplot altogether, which will give you a little more control, by calling the API directly as shown in api example code: agg_oo.py.

You can either generate hardcopy on the filesystem by calling savefig:

```
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot([1,2,3])
fig.savefig('test.png')
```

or by saving to a file handle:

```
import sys
fig.savefig(sys.stdout)
```

Here is an example using Pillow. First, the figure is saved to a StringIO object which is then fed to Pillow for further processing:

```
import StringIO, Image
imgdata = StringIO.StringIO()
fig.savefig(imgdata, format='png')
```
imgdata.seek(0)  # rewind the data
im = Image.open(imgdata)

11.3.1 matplotlib with apache

TODO; see Contribute to matplotlib documentation.

11.3.2 matplotlib with django

TODO; see Contribute to matplotlib documentation.

11.3.3 matplotlib with zope

TODO; see Contribute to matplotlib documentation.

11.3.4 Clickable images for HTML

Andrew Dalke of Dalke Scientific has written a nice article on how to make html click maps with matplotlib agg PNGs. We would also like to add this functionality to SVG. If you are interested in contributing to these efforts that would be great.

11.4 Search examples

The nearly 300 code Matplotlib Examples included with the matplotlib source distribution are full-text searchable from the search page, but sometimes when you search, you get a lot of results from the The Matplotlib API or other documentation that you may not be interested in if you just want to find a complete, free-standing, working piece of example code. To facilitate example searches, we have tagged every code example page with the keyword codex for code example which shouldn’t appear anywhere else on this site except in the FAQ. So if you want to search for an example that uses an ellipse, search for codex ellipse.

11.5 Cite Matplotlib

If you want to refer to matplotlib in a publication, you can use “Matplotlib: A 2D Graphics Environment” by J. D. Hunter In Computing in Science & Engineering, Vol. 9, No. 3. (2007), pp. 90-95 (see this reference page):

@article{Hunter:2007,
    Address = {10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1314 USA},
    Author = {Hunter, John D.},
    Date-Added = {2010-09-23 12:22:10 -0700},
    Date-Modified = {2010-09-23 12:22:10 -0700},
    Isi = {000245668100019},
    Isi-Recid = {155389429},
}
Abstract = {Matplotlib is a 2D graphics package used for Python for application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems.},
12.1 Obtaining matplotlib version

To find out your matplotlib version number, import it and print the `__version__` attribute:

```python
>>> import matplotlib
>>> matplotlib.__version__
'0.98.0'
```

12.2 matplotlib install location

You can find what directory matplotlib is installed in by importing it and printing the `__file__` attribute:

```python
>>> import matplotlib
>>> matplotlib.__file__
'/home/jdhunter/dev/lib64/python2.5/site-packages/matplotlib/__init__.pyc'
```

12.3 .matplotlib directory location

Each user has a matplotlib configuration directory which may contain a `matplotlibrc` file. To locate your `.matplotlib/` directory, use `matplotlib.get_configdir()`:
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'/home/darren/.matplotlib'

On unix-like systems, this directory is generally located in your HOME directory. On windows, it is in your documents and settings directory by default:

>>> import matplotlib
>>> mpl.get_configdir()  
'C:\Documents and Settings\jdhunter\.matplotlib'

If you would like to use a different configuration directory, you can do so by specifying the location in your MPLCONFIGDIR environment variable – see Setting environment variables in Linux and OS-X.

12.4 Getting help

There are a number of good resources for getting help with matplotlib. There is a good chance your question has already been asked:

- The mailing list archive.
- Github issues.
- Stackoverflow questions tagged matplotlib.

If you are unable to find an answer to your question through search, please provide the following information in your e-mail to the mailing list:

- your operating system; (Linux/UNIX users: post the output of `uname -a`)
- matplotlib version:
  python -c `import matplotlib; print matplotlib.__version__`
- where you obtained matplotlib (e.g., your Linux distribution’s packages or the matplotlib Sourceforge site, or Anaconda or Enthought Canopy).
- any customizations to your matplotlibrc file (see Customizing matplotlib).
- if the problem is reproducible, please try to provide a minimal, standalone Python script that demonstrates the problem. This is the critical step. If you can’t post a piece of code that we can run and reproduce your error, the chances of getting help are significantly diminished. Very often, the mere act of trying to minimize your code to the smallest bit that produces the error will help you find a bug in your code that is causing the problem.
- you can get very helpful debugging output from matplotlib by running your script with a verbose-helpful or --verbose-debug flags and posting the verbose output the lists:
  
  > python simple_plot.py --verbose-helpful > output.txt

If you compiled matplotlib yourself, please also provide

- any changes you have made to setup.py or setupext.py
• the output of:

    rm -rf build
    python setup.py build

    The beginning of the build output contains lots of details about your platform that are useful for the matplotlib developers to diagnose your problem.

• your compiler version – e.g., gcc --version

Including this information in your first e-mail to the mailing list will save a lot of time.

You will likely get a faster response writing to the mailing list than filing a bug in the bug tracker. Most developers check the bug tracker only periodically. If your problem has been determined to be a bug and can not be quickly solved, you may be asked to file a bug in the tracker so the issue doesn’t get lost.

12.5 Problems with recent git versions

First make sure you have a clean build and install (see How to completely remove matplotlib), get the latest git update, install it and run a simple test script in debug mode:

    rm -rf build
    rm -rf /path/to/site-packages/matplotlib*
    git pull
    python setup.py install > build.out
    python examples/pylab_examples/simple_plot.py --verbose-debug > run.out

and post build.out and run.out to the matplotlib-devel mailing list (please do not post git problems to the users list).

Of course, you will want to clearly describe your problem, what you are expecting and what you are getting, but often a clean build and install will help. See also Getting help.
CHAPTER
THIRTEEN

ENVIRONMENT VARIABLES

Contents

- Environment Variables
  - Setting environment variables in Linux and OS-X
    - BASH/KSH
    - CSH/TCSH
  - Setting environment variables in windows

HOME
The user’s home directory. On linux, ~ is shorthand for HOME.

PATH
The list of directories searched to find executable programs

PYTHONPATH
The list of directories that is added to Python’s standard search list when importing packages and modules

MPLCONFIGDIR
This is the directory used to store user customizations to matplotlib, as well as some caches to improve performance. If MPLCONFIGDIR is not defined, HOME/.matplotlib is used if it is writable. Otherwise, the python standard library tempfile.gettempdir() is used to find a base directory in which the matplotlib subdirectory is created.

13.1 Setting environment variables in Linux and OS-X

To list the current value of PYTHONPATH, which may be empty, try:

echo $PYTHONPATH

The procedure for setting environment variables in depends on what your default shell is. BASH seems to be the most common, but CSH is also common. You should be able to determine which by running at the command prompt:

echo $SHELL
13.1.1 BASH/KSH

To create a new environment variable:

```bash
export PYTHONPATH=~/Python
```

To prepend to an existing environment variable:

```bash
export PATH=~/bin:${PATH}
```

The search order may be important to you, do you want ~/bin to be searched first or last? To append to an existing environment variable:

```bash
export PATH=${PATH}:~/bin
```

To make your changes available in the future, add the commands to your ~/.bashrc file.

13.1.2 CSH/TCSH

To create a new environment variable:

```bash
setenv PYTHONPATH ~/Python
```

To prepend to an existing environment variable:

```bash
setenv PATH ~/bin:${PATH}
```

The search order may be important to you, do you want ~/bin to be searched first or last? To append to an existing environment variable:

```bash
setenv PATH ${PATH}:~/bin
```

To make your changes available in the future, add the commands to your ~/.cshrc file.

13.2 Setting environment variables in windows

Open the Control Panel (Start → Control Panel), start the System program. Click the Advanced tab and select the Environment Variables button. You can edit or add to the User Variables.
Part III

External Resources
BOOKS AND CHAPTERS

- Matplotlib for Python Developers by Sandro Tosi
- Matplotlib chapter by John Hunter and Michael Droettboom in The Architecture of Open Source Applications
- Graphics with Matplotlib by David J. Raymond
• Getting started with Matplotlib by unpingco
• Plotting with matplotlib by Mike Müller
• Introduction to NumPy and Matplotlib by Eric Jones
• Anatomy of Matplotlib by Benjamin Root
TUTORIALS

- Matplotlib tutorial by Nicolas P. Rougier
- Anatomy of Matplotlib - IPython Notebooks by Benjamin Root
Part IV

The Matplotlib Developers’ Guide
17.1 Pull request checklist

This checklist should be consulted when creating pull requests to make sure they are complete before merging. These are not intended to be rigidly followed—it’s just an attempt to list in one place all of the items that are necessary for a good pull request. Of course, some items will not always apply.

17.1.1 Branch selection

- In general, simple bugfixes that are unlikely to introduce new bugs of their own should be merged onto the maintenance branch. New features, or anything that changes the API, should be made against master. The rules are fuzzy here – when in doubt, try to get some consensus.
  - Once changes are merged into the maintenance branch, they should be merged into master.

17.1.2 Style

- Formatting should follow PEP8. Exceptions to these rules are acceptable if it makes the code objectively more readable.
  - You should consider installing/enabling automatic PEP8 checking in your editor. Part of the test suite is checking PEP8 compliance, things go smoother if the code is mostly PEP8 compliant to begin with.

- No tabs (only spaces). No trailing whitespace.
  - Configuring your editor to remove these things upon saving will save a lot of trouble.

- Import the following modules using the standard scipy conventions:

```python
import numpy as np
import numpy.ma as ma
import matplotlib as mpl
from matplotlib import pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.collections as mcol
import matplotlib.patches as mpatches
```
• See below for additional points about *Keyword argument processing*, if code in your pull request does that.

• Adding a new pyplot function involves generating code. See *Writing a new pyplot function* for more information.

**17.1.3 Documentation**

• Every new feature should be documented. If it’s a new module, don’t forget to add a new rst file to the API docs.

• Docstrings should be in *numpydoc format*. Don’t be thrown off by the fact that many of the existing docstrings are not in that format; we are working to standardize on numpydoc.

Docstrings should look like (at a minimum):

```python
def foo(bar, baz=None):
    
    This is a prose description of foo and all the great things it does.

    Parameters
    ----------
    bar : (type of bar)
        A description of bar

    baz : (type of baz), optional
        A description of baz

    Returns
    -------
    foobar : (type of foobar)
        A description of foobar
    foobaz : (type of foobaz)
        A description of foobaz

    # some very clever code
    return foobar, foobaz
```

• Each high-level plotting function should have a simple example in the Example section of the docstring. This should be as simple as possible to demonstrate the method. More complex examples should go in the examples tree.

• Build the docs and make sure all formatting warnings are addressed.

• See *Documenting matplotlib* for our documentation style guide.

• If your changes are non-trivial, please make an entry in the CHangelog.

• If your change is a major new feature, add an entry to doc/users/whats_new.rst.

• If you change the API in a backward-incompatible way, please document it in doc/api/api_changes.rst.
17.1.4 Testing

Using the test framework is discussed in detail in the section Testing.

- If the PR is a bugfix, add a test that fails prior to the change and passes with the change. Include any relevant issue numbers in the docstring of the test.
- If this is a new feature, add a test that exercises as much of the new feature as possible. (The --with-coverage option may be useful here).
- Make sure the Travis tests are passing before merging.
  - The Travis tests automatically test on all of the Python versions matplotlib supports whenever a pull request is created or updated. The tox support in matplotlib may be useful for testing locally.

17.1.5 Installation

- If you have added new files or directories, or reorganized existing ones, make sure the new files included in the match patterns in MANIFEST.in, and/or in package_data in setup.py.

17.1.6 C/C++ extensions

- Extensions may be written in C or C++.
- Code style should conform to PEP7 (understanding that PEP7 doesn’t address C++, but most of its admonitions still apply).
- Interfacing with Python may be done either with the raw Python/C API or Cython. Use of PyCXX is discouraged for new code.
- Python/C interface code should be kept separate from the core C/C++ code. The interface code should be named FOO_wrap.cpp.
- Header file documentation (aka docstrings) should be in Numpydoc format. We don’t plan on using automated tools for these docstrings, and the Numpydoc format is well understood in the scientific Python community.

17.2 Style guide

17.2.1 Keyword argument processing

Matplotlib makes extensive use of **kwargs for pass-through customizations from one function to another. A typical example is in matplotlib.pyplot.text(). The definition of the pyplot text function is a simple pass-through to matplotlib.axes.Axes.text():

```python
# in pylab.py
def text(*args, **kwargs):
    ret = gca().text(*args, **kwargs)
```
draw_if_interactive()
return ret

text() in simplified form looks like this, i.e., it just passes all args and kwargs on to matplotlib.text.Text.__init__():

# in axes.py
def text(self, x, y, s, fontdict=None, withdash=False, **kwargs):
t = Text(x=x, y=y, text=s, **kwargs)

and __init__() (again with liberties for illustration) just passes them on to the matplotlib.artist.Artist.update() method:

# in text.py
def __init__(self, x=0, y=0, text='', **kwargs):
    Artist.__init__(self)
    self.update(kwargs)

update does the work looking for methods named like set_property if property is a keyword argument. i.e., no one looks at the keywords, they just get passed through the API to the artist constructor which looks for suitably named methods and calls them with the value.

As a general rule, the use of **kwargs should be reserved for pass-through keyword arguments, as in the example above. If all the keyword args are to be used in the function, and not passed on, use the key/value keyword args in the function definition rather than the **kwargs idiom.

In some cases, you may want to consume some keys in the local function, and let others pass through. You can pop the ones to be used locally and pass on the rest. For example, in plot(), scalex and scaley are local arguments and the rest are passed on as Line2D() keyword arguments:

# in axes.py
def plot(self, *args, **kwargs):
scalex = kwargs.pop('scalex', True)
scaley = kwargs.pop('scaley', True)
if not self._hold: self.cla()
lines = []
for line in self._get_lines(*args, **kwargs):
    self.add_line(line)
    lines.append(line)

Note: there is a use case when kwargs are meant to be used locally in the function (not passed on), but you still need the **kwargs idiom. That is when you want to use *args to allow variable numbers of non-keyword args. In this case, python will not allow you to use named keyword args after the *args usage, so you will be forced to use **kwargs. An example is matplotlib.contour.ContourLabeler.clabel():

# in contour.py
def clabel(self, *args, **kwargs):
    fontsize = kwags.get('fontsize', None)
    inline = kwags.get('inline', 1)
    self.fmt = kwags.get('fmt', '%1.3f')
    colors = kwags.get('colors', None)
    if len(args) == 0:
        levels = self.levels
indices = range(len(self.levels))

elif len(args) == 1:
    ...etc...

17.3 Hints

This section describes how to add certain kinds of new features to matplotlib.

17.3.1 Developing a new backend

If you are working on a custom backend, the backend setting in matplotlibrc (Customizing matplotlib) supports an external backend via the module directive. If my_backend.py is a matplotlib backend in your PYTHONPATH, you can set use it on one of several ways

- in matplotlibrc:
  backend : module://my_backend

- with the use directive in your script:

  ```python
  import matplotlib
  matplotlib.use('module://my_backend')
  ```

- from the command shell with the -d flag:

  ```bash
  > python simple_plot.py -d module://my_backend
  ```

17.3.2 Writing examples

We have hundreds of examples in subdirectories of matplotlib/examples, and these are automatically generated when the website is built to show up both in the examples and gallery sections of the website.

Any sample data that the example uses should be kept small and distributed with matplotlib in the lib/matplotlib/mpl-data/sample_data/ directory. Then in your example code you can load it into a file handle with:

```python
import matplotlib.cbook as cbook
fh = cbook.get_sample_data('mydata.dat')
```

17.3.3 Writing a new pyplot function

A large portion of the pyplot interface is automatically generated by the boilerplate.py script (in the root of the source tree). To add or remove a plotting method from pyplot, edit the appropriate list in boilerplate.py and then run the script which will update the content in lib/matplotlib/pyplot.py. Both the changes in boilerplate.py and lib/matplotlib/pyplot.py should be checked into the repository.
Note: boilerplate.py looks for changes in the installed version of matplotlib and not the source tree. If you expect the pyplot.py file to show your new changes, but they are missing, this might be the cause.

Install your new files by running python setup.py build and python setup.py install followed by python boilerplate.py. The new pyplot.py file should now have the latest changes.
As of matplotlib 1.4, the six library is used to support Python 2 and 3 from a single code base. The 2to3 tool is no longer used.

This document describes some of the issues with that approach and some recommended solutions. It is not a complete guide to Python 2 and 3 compatibility.

18.1 Welcome to the __future__

The top of every py file should include the following:

```python
from __future__ import (absolute_import, division, print_function, unicode_literals)
import six
```

This will make the Python 2 interpreter behave as close to Python 3 as possible.

All matplotlib files should also import six, whether they are using it or not, just to make moving code between modules easier, as six gets used a lot.

18.2 Finding places to use six

The only way to make sure code works on both Python 2 and 3 is to make sure it is covered by unit tests.

However, the 2to3 commandline tool can also be used to locate places that require special handling with six.

(The modernize tool may also be handy, though I’ve never used it personally).

The six documentation serves as a good reference for the sorts of things that need to be updated.

18.3 The dreaded \u escapes

When from __future__ import unicode_literals is used, all string literals (not preceded with a b) will become unicode literals.
Normally, one would use “raw” string literals to encode strings that contain a lot of slashes that we don’t want Python to interpret as special characters. A common example in matplotlib is when it deals with TeX and has to represent things like r"\usepackage{foo}". Unfortunately, on Python 2 there is no way to represent u in a raw unicode string literal, since it will always be interpreted as the start of a unicode character escape, such as u20af. The only solution is to use a regular (non-raw) string literal and repeat all slashes, e.g. "\usepackage{foo}".

The following shows the problem on Python 2:

```python
>>> ur'\u'
    File "<stdin>", line 1
SyntaxError: (unicode error) 'rawunicodeescape' codec can't decode bytes in position 0-1: truncated \uXXXX
>>> ur'\\u'
u'\\\u'
>>> u'\\u'
    File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 0-1: truncated \uXXXX escape
>>> u'\\u'
u'\\u'
```

This bug has been fixed in Python 3, however, we can’t take advantage of that and still support Python 2:

```python
>>> r'\u'
'\\u'
>>> r'\\u'
'\\\\u'
>>> '\u'
    File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 0-1: truncated \uXXXX escape
>>> '\u'
'\\u'
```

18.4 Iteration

The behavior of the methods for iterating over the items, values and keys of a dictionary has changed in Python 3. Additionally, other built-in functions such as zip, range and map have changed to return iterators rather than temporary lists.

In many cases, the performance implications of iterating vs. creating a temporary list won’t matter, so it’s tempting to use the form that is simplest to read. However, that results in code that behaves differently on Python 2 and 3, leading to subtle bugs that may not be detected by the regression tests. Therefore, unless the loop in question is provably simple and doesn’t call into other code, the six versions that ensure the same behavior on both Python 2 and 3 should be used. The following table shows the mapping of equivalent semantics between Python 2, 3 and six for dict.items():

<table>
<thead>
<tr>
<th>Python 2</th>
<th>Python 3</th>
<th>six</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.items()</td>
<td>list(d.items())</td>
<td>list(six.iteritems(d))</td>
</tr>
<tr>
<td>d.iteritems()</td>
<td>d.items()</td>
<td>six.iteritems(d)</td>
</tr>
</tbody>
</table>
18.5 Numpy-specific things

When specifying dtypes, all strings must be byte strings on Python 2 and unicode strings on Python 3. The best way to handle this is to force cast them using `str()`. The same is true of structure specifiers in the `struct` built-in module.
Matplotlib only uses BSD compatible code. If you bring in code from another project make sure it has a PSF, BSD, MIT or compatible license (see the Open Source Initiative licenses page for details on individual licenses). If it doesn’t, you may consider contacting the author and asking them to relicense it. GPL and LGPL code are not acceptable in the main code base, though we are considering an alternative way of distributing L/GPL code through an separate channel, possibly a toolkit. If you include code, make sure you include a copy of that code’s license in the license directory if the code’s license requires you to distribute the license with it. Non-BSD compatible licenses are acceptable in matplotlib toolkits (e.g., basemap), but make sure you clearly state the licenses you are using.

19.1 Why BSD compatible?

The two dominant license variants in the wild are GPL-style and BSD-style. There are countless other licenses that place specific restrictions on code reuse, but there is an important difference to be considered in the GPL and BSD variants. The best known and perhaps most widely used license is the GPL, which in addition to granting you full rights to the source code including redistribution, carries with it an extra obligation. If you use GPL code in your own code, or link with it, your product must be released under a GPL compatible license. i.e., you are required to give the source code to other people and give them the right to redistribute it as well. Many of the most famous and widely used open source projects are released under the GPL, including linux, gcc, emacs and sage.

The second major class are the BSD-style licenses (which includes MIT and the python PSF license). These basically allow you to do whatever you want with the code: ignore it, include it in your own open source project, include it in your proprietary product, sell it, whatever. python itself is released under a BSD compatible license, in the sense that, quoting from the PSF license page:

There is no GPL-like "copyleft" restriction. Distributing binary-only versions of Python, modified or not, is allowed. There is no requirement to release any of your source code. You can also write extension modules for Python and provide them only in binary form.

Famous projects released under a BSD-style license in the permissive sense of the last paragraph are the BSD operating system, python and TeX.

There are several reasons why early matplotlib developers selected a BSD compatible license. matplotlib is a python extension, and we choose a license that was based on the python license (BSD compatible). Also, we wanted to attract as many users and developers as possible, and many software companies will
not use GPL code in software they plan to distribute, even those that are highly committed to open source development, such as enthought, out of legitimate concern that use of the GPL will “infect” their code base by its viral nature. In effect, they want to retain the right to release some proprietary code. Companies and institutions who use matplotlib often make significant contributions, because they have the resources to get a job done, even a boring one. Two of the matplotlib backends (FLTK and WX) were contributed by private companies. The final reason behind the licensing choice is compatibility with the other python extensions for scientific computing: ipython, numpy, scipy, the enthought tool suite and python itself are all distributed under BSD compatible licenses.
WORKING WITH *MATPLOTLIB* SOURCE CODE

Contents:

20.1 Introduction

These pages describe a *git* and *github* workflow for the *matplotlib* project. There are several different workflows here, for different ways of working with *matplotlib*. This is not a comprehensive *git* reference, it’s just a workflow for our own project. It’s tailored to the *github* hosting service. You may well find better or quicker ways of getting stuff done with *git*, but these should get you started.

For general resources for learning *git* see *git resources*.

20.2 Install *git*

20.2.1 Overview

<table>
<thead>
<tr>
<th>Platform</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debian / Ubuntu</td>
<td><code>sudo apt-get install git-core</code></td>
</tr>
<tr>
<td>Fedora</td>
<td><code>sudo yum install git-core</code></td>
</tr>
<tr>
<td>Windows</td>
<td>Download and install <em>msysGit</em></td>
</tr>
<tr>
<td>OS X</td>
<td>Use the <em>git-osx-installer</em></td>
</tr>
</tbody>
</table>

20.2.2 In detail

See the *git* page for the most recent information.

Have a look at the *github* install help pages available from *github help*

There are good instructions here: [http://book.git-scm.com/2_installing_git.html](http://book.git-scm.com/2_installing_git.html)
20.3 Following the latest source

These are the instructions if you just want to follow the latest matplotlib source, but you don’t need to do any development for now.

The steps are:

- Install git
- get local copy of the git repository from github
- update local copy from time to time

20.3.1 Get the local copy of the code

From the command line:
```
git clone git://github.com/matplotlib/matplotlib.git
```

You now have a copy of the code tree in the new matplotlib directory.

20.3.2 Updating the code

From time to time you may want to pull down the latest code. Do this with:
```
cd matplotlib
git pull
```

The tree in matplotlib will now have the latest changes from the initial repository.

20.4 Git for development

Contents:

20.4.1 Making your own copy (fork) of matplotlib

You need to do this only once. The instructions here are very similar to the instructions at http://help.github.com/forking/ — please see that page for more detail. We’re repeating some of it here just to give the specifics for the matplotlib project, and to suggest some default names.

Set up and configure a github account

If you don’t have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the Generating SSH keys help on github help.
Create your own forked copy of matplotlib

1. Log into your github account.
2. Go to the matplotlib github home at matplotlib github.
3. Click on the fork button:

Now, after a short pause you should find yourself at the home page for your own forked copy of matplotlib.

20.4.2 Set up your fork

First you follow the instructions for Making your own copy (fork) of matplotlib.

Overview

git clone git@github.com:your-user-name/matplotlib.git
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git

In detail

Clone your fork

1. Clone your fork to the local computer with git clone git@github.com:your-user-name/matplotlib.git
2. Investigate. Change directory to your new repo: cd matplotlib. Then git branch -a to show you all branches. You'll get something like:

* master
remotes/origin/master

This tells you that you are currently on the master branch, and that you also have a remote connection to origin/master. What remote repository is remote/origin? Try git remote -v to see the URLs for the remote. They will point to your github fork.

Now you want to connect to the upstream matplotlib github repository, so you can merge in changes from trunk.

20.4. Git for development
Linking your repository to the upstream repo

```
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git
```

`upstream` here is just the arbitrary name we’re using to refer to the main `matplotlib` repository at `matplotlib github`.

Note that we’ve used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can’t accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Note this command needs to be run on every clone of the repository that you make. It is not tracked in your personal repository on `github`.

Just for your own satisfaction, show yourself that you now have a new ‘remote’, with `git remote -v` show, giving you something like:

```
upstream git://github.com/matplotlib/matplotlib.git (fetch)
upstream git://github.com/matplotlib/matplotlib.git (push)
origin git@github.com:your-user-name/matplotlib.git (fetch)
origin git@github.com:your-user-name/matplotlib.git (push)
```

### 20.4.3 Configure git

**Overview**

Your personal `git` configurations are saved in the `.gitconfig` file in your home directory. Here is an example `.gitconfig` file:

```
[user]
    name = Your Name
    email = you@yourdomain.example.com

[alias]
    ci = commit -a
    co = checkout
    st = status -a
    stat = status -a
    br = branch
    wdiff = diff --color-words

[core]
    editor = vim

[merge]
    summary = true

[apply]
    whitespace = fix
```
[core]
    autocrlf = input

You can edit this file directly or you can use the `git config --global` command:

```bash
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

### In detail

#### user.name and user.email

It is good practice to tell `git` who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```bash
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```bash
[user]
    name = Your Name
    email = you@yourdomain.example.com
```

Of course you’ll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

#### Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```bash
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
```
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"

will create an alias section in your .gitconfig file with contents like this:

```
[alias]
   ci = commit -a
   co = checkout
   st = status -a
   stat = status -a
   br = branch
   wdiff = diff --color-words
```

**Editor**

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

**Merging**

To enforce summaries when doing merges (~/.gitconfig file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

### 20.4.4 Development workflow

You’ve discovered a bug or something else you want to change in matplotlib .. — excellent!

You’ve worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way to contribute to matplotlib is through github. If for some reason you don’t want to use github, see Making patches for instructions on how to email patches to the mailing list.

You already have your own forked copy of the matplotlib repository, by following Making your own copy (fork) of matplotlib, Set up your fork, and you have configured git by following Configure git.

**Workflow summary**

- Keep your master branch clean of edits that have not been merged to the main matplotlib development repo. Your master then will follow the main matplotlib repository.
• Start a new feature branch for each set of edits that you do.

• Do not merge the master branch or maintenance tracking branches into your feature branch. If you need to include commits from upstream branches (either to pick up a bug fix or to resolve a conflict) please rebase your branch on the upstream branch.

• Ask for review!

This way of working really helps to keep work well organized, and in keeping history as clear as possible. See — for example — linux git workflow.

**Making a new feature branch**

git checkout -b my-new-feature master

This will create and immediately check out a feature branch based on master. To create a feature branch based on a maintenance branch, use:

git fetch origin
git checkout -b my-new-feature origin/v1.0.x

Generally, you will want to keep this also on your public github fork of matplotlib. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called origin. You push up to your own repo on github with:

git push origin my-new-feature

You will need to use this exact command, rather than simply git push every time you want to push changes on your feature branch to your github repo. However, in git >1.7 you can set up a link by using the --set-upstream option:

git push --set-upstream origin my-new-feature

and then next time you need to push changes to your branch a simple git push will suffice. Note that git push pushes out all branches that are linked to a remote branch.

**The editing workflow**

**Overview**

```bash
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

**In more detail**

1. Make some changes
2. See which files have changed with `git status` (see `git status`). You’ll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
# modified: README
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
# INSTALL
```

3. Check what the actual changes are with `git diff` (git diff).

4. Add any new files to version control `git add new_file_name` (see `git add`).

5. To commit all modified files into the local copy of your repo, do `git commit -am 'A commit message'`. Note the `-am` options to commit. The `-m` flag just signals that you’re going to type a message on the command line. The `-a` flag — you can just take on faith — or see why the `-a` flag? — and the helpful use-case description in the tangled working copy problem. The `git commit` manual page might also be useful.

6. To push the changes up to your forked repo on github, do a `git push` (see `git push`).

**Asking for code review — open a Pull Request (PR)**

It’s a good idea to consult the `Pull request checklist` to make sure your pull request is ready for merging.

1. Go to your repo URL — e.g., `http://github.com/your-user-name/matplotlib`.
2. Select your feature branch from the drop down menu:
3. Click on the green button:
4. Make sure that you are requesting a pull against the correct branch
5. Enter a PR heading and description (if there is only one commit in the PR github will automatically fill these fields for you). If this PR is addressing a specific issue, please reference it by number (ex #1325) which github will automatically make into links.
6. Click ‘Create Pull Request’ button!
7. Discussion of the change will take place in the pull request thread.

**Staying up to date with changes in the central repository**

This updates your working copy from the upstream `matplotlib` github repo.
Overview

# go to your master branch
git checkout master
# pull changes from github
git fetch upstream
# merge from upstream
git merge --ff-only upstream/master

In detail

We suggest that you do this only for your master branch, and leave your ‘feature’ branches unmerged, to keep their history as clean as possible. This makes code review easier:

git checkout master

Make sure you have done Linking your repository to the upstream repo.

Merge the upstream code into your current development by first pulling the upstream repo to a copy on your local machine:

git fetch upstream

then merging into your current branch:

git merge --ff-only upstream/master

The --ff-only option guarantees that if you have mistakenly committed code on your master branch, the merge fails at this point. If you were to merge upstream/master to your master, you would start to diverge from the upstream. If this command fails, see the section on accidents.

The letters ‘ff’ in --ff-only mean ‘fast forward’, which is a special case of merge where git can simply update your branch to point to the other branch and not do any actual merging of files. For master and other integration branches this is exactly what you want.

Other integration branches

Some people like to keep separate local branches corresponding to the maintenance branches on github. At the time of this writing, v1.0.x is the active maintenance branch. If you have such a local branch, treat is just as master: don’t commit on it, and before starting new branches off of it, update it from upstream:

git checkout v1.0.x
git fetch upstream
git merge --ff-only upstream/v1.0.x

But you don’t necessarily have to have such a branch. Instead, if you are preparing a bugfix that applies to the maintenance branch, fetch from upstream and base your bugfix on the remote branch:

git fetch upstream
git checkout -b my-bug-fix upstream/v1.0.x
Recovering from accidental commits on master

If you have accidentally committed changes on master and `git merge --ff-only` fails, don’t panic! First find out how much you have diverged:

```
    git diff upstream/master...master
```

If you find that you want simply to get rid of the changes, reset your master branch to the upstream version:

```
    git reset --hard upstream/master
```

As you might surmise from the words ‘reset’ and ‘hard’, this command actually causes your changes to the current branch to be lost, so think twice.

If, on the other hand, you find that you want to preserve the changes, create a feature branch for them:

```
    git checkout -b my-important-changes
```

Now `my-important-changes` points to the branch that has your changes, and you can safely reset master as above — but make sure to reset the correct branch:

```
    git checkout master
    git reset --hard upstream/master
```

Deleting a branch on github

```
    git checkout master
    # delete branch locally
    git branch -D my-unwanted-branch
    # delete branch on github
    git push origin :my-unwanted-branch
```

(Note the colon : before test-branch. See also: http://github.com/guides/remove-a-remote-branch)

Exploring your repository

To see a graphical representation of the repository branches and commits:

```
    gitk --all
```

To see a linear list of commits for this branch:

```
    git log
```

You can also look at the network graph visualizer for your github repo.

20.4.5 Two and three dots in difference specs

Thanks to Yarik Halchenko for this explanation.
Imagine a series of commits A, B, C, D... Imagine that there are two branches, topic and master. You branched topic off master when master was at commit ‘E’. The graph of the commits looks like this:

```
A---B---C topic
/  
D---E---F---G master
```

Then:

```
git diff master..topic
```

will output the difference from G to C (i.e. with effects of F and G), while:

```
git diff master...topic
```

would output just differences in the topic branch (i.e. only A, B, and C).

### 20.5 git resources

#### 20.5.1 Tutorials and summaries

- [github help](https://help.github.com) has an excellent series of how-to guides.
- [learn.github](https://learn.github.com) has an excellent series of tutorials
- A [git cheat sheet](https://git-scm.com/docs) is a page giving summaries of common commands.
- The [git user manual](https://git-scm.com/docs)
- The [git tutorial](https://git-scm.com/book)
- The [git community book](https://git-scm.com/docs)
- [git ready](https://git-scm.com/docs) — a nice series of tutorials
- [git casts](https://git-casts.com) — video snippets giving git how-tos.
- [git magic](https://git-scm.com/docs) — extended introduction with intermediate detail
- The [git parable](https://git-scm.com/docs) is an easy read explaining the concepts behind git.
- Our own [git foundation](https://git-scm.com/docs) expands on the [git parable](https://git-scm.com/docs).
- [Fernando Perez’ git page](https://fernando.perez.org/git/) — [Fernando’s git page](https://fernando.perez.org/git/) — many links and tips
- A good but technical page on [git concepts](https://git-scm.com/docs)
- [git svn crash course](https://git-scm.com/docs): git for those of us used to [subversion](https://subversion.apache.org/)

#### 20.5.2 Advanced git workflow

There are many ways of working with [git](https); here are some posts on the rules of thumb that other projects have come up with:
Linus Torvalds on git management

Linus Torvalds on linux git workflow. Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

### 20.5.3 Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- `git add`
- `git branch`
- `git checkout`
- `git clone`
- `git commit`
- `git config`
- `git diff`
- `git log`
- `git pull`
- `git push`
- `git remote`
- `git status`

### 20.6 Making a patch

#### 20.6.1 Making patches

**Overview**

```bash
# tell git who you are
git config --global user.email you@yourdomain.example.com
# get the repository if you don't have it
git clone git://github.com/matplotlib/matplotlib.git
# make a branch for your patching
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
```

Chapter 20. Working with matplotlib source code
# commit work in progress as you go

```bash
git commit -am 'BF - added tests for Funny bug'
```

# hack hack, hack

```bash
git commit -am 'BF - added fix for Funny bug'
```

# make the patch files

```bash
git format-patch -M -C master
```

Then, send the generated patch files to the matplotlib mailing list — where we will thank you warmly.

## In detail

1. Tell git who you are so it can label the commits you’ve made:

   ```bash
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don’t already have one, clone a copy of the matplotlib repository:

   ```bash
git clone git://github.com/matplotlib/matplotlib.git
cd matplotlib
```

3. Make a ‘feature branch’. This will be where you work on your bug fix. It’s nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

   ```bash
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

   ```bash
# hack, hack, hack
# Tell git about any new files you’ve made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

   Note the -am options to commit. The m flag just signals that you’re going to type a message on the command line. The a flag — you can just take on faith — or see why the -a flag?

5. When you have finished, check you have committed all your changes:

   ```bash
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the master branch:

   ```bash
git format-patch -M -C master
```

You will now have several files named for the commits:

- 0001-BF-added-tests-for-Funny-bug.patch
- 0002-BF-added-fix-for-Funny-bug.patch

---

20.6. Making a patch 419
Send these files to the matplotlib mailing list.

When you are done, to switch back to the main copy of the code, just return to the master branch:

git checkout master
Matplotlib has a testing infrastructure based on nose, making it easy to write new tests. The tests are in `matplotlib.tests`, and customizations to the nose testing infrastructure are in `matplotlib.testing`. (There is other old testing cruft around, please ignore it while we consolidate our testing to these locations.)

### 21.1 Requirements

The following software is required to run the tests:

- **nose**, version 1.0 or later
- **Ghostscript** (to render PDF files)
- **Inkscape** (to render SVG files)

### 21.2 Running the tests

Running the tests is simple. Make sure you have nose installed and run the script `tests.py` in the root directory of the distribution. The script can take any of the usual `nose` test arguments, such as

```
-v
-d
--with-coverage
```

Increase verbosity, detailed error messages, and enable collecting coverage information.

To run a single test from the command line, you can provide a dot-separated path to the module followed by the function separated by a colon, e.g., (this is assuming the test is installed):

```
python tests.py matplotlib.tests.test_simplification:test_clipping
```

If you want to run the full test suite, but want to save wall time try running the tests in parallel:

```
python ../matplotlib/tests.py -sv --processes=5 --process-timeout=300
```

as we do on Travis.ci.

An alternative implementation that does not look at command line arguments works from within Python:
import matplotlib
matplotlib.test()

Running tests by any means other than matplotlib.test() does not load the nose “knownfailureif” (Known failing tests) plugin, causing known-failing tests to fail for real.

### 21.3 Writing a simple test

Many elements of Matplotlib can be tested using standard tests. For example, here is a test from matplotlib.tests.test_basic:

```python
from nose.tools import assert_equal
def test_simple():
    """
    very simple example test
    """
    assert_equal(1+1, 2)
```

Nose determines which functions are tests by searching for functions beginning with “test” in their name. If the test has side effects that need to be cleaned up, such as creating figures using the pyplot interface, use the @cleanup decorator:

```python
from matplotlib.testing.decorators import cleanup
def test_create_figure():
    """
    very simple example test that creates a figure using pyplot.
    """
    fig = figure()
    ...
```

### 21.4 Writing an image comparison test

Writing an image based test is only slightly more difficult than a simple test. The main consideration is that you must specify the “baseline”, or expected, images in the image_comparison() decorator. For example, this test generates a single image and automatically tests it:

```python
import numpy as np
import matplotlib
from matplotlib.testing.decorators import image_comparison
import matplotlib.pyplot as plt

@image_comparison(baseline_images=['spines_axes_positions'])
def test_spines_axes_positions():
    # SF bug 2852168
    fig = plt.figure()
```
```python
x = np.linspace(0,2*np.pi,100)
y = 2*np.sin(x)
ax = fig.add_subplot(1,1,1)
ax.set_title('centered spines')
ax.plot(x,y)
ax.spines['right'].set_position(('axes',0.1))
ax.yaxis.set_ticks_position('right')
ax.spines['top'].set_position(('axes',0.25))
ax.xaxis.set_ticks_position('top')
ax.spines['left'].set_color('none')
ax.spines['bottom'].set_color('none')
```

The first time this test is run, there will be no baseline image to compare against, so the test will fail. Copy the output images (in this case result_images/test_category/spines_axes_positions.*) to the correct subdirectory of baseline_images tree in the source directory (in this case lib/matplotlib/tests/baseline_images/test_category). Note carefully the * at the end: this will copy only the images we need to include in the git repository. The files ending in _pdf.png and _svg.png are converted from the pdf and svg originals on the fly and do not need to be in the repository. Put these new files under source code revision control (with git add). When rerunning the tests, they should now pass.

There are two optional keyword arguments to the image_comparison decorator:

- **extensions**: If you only wish to test some of the image formats (rather than the default png, svg and pdf formats), pass a list of the extensions to test.
- **tol**: This is the image matching tolerance, the default 1e-3. If some variation is expected in the image between runs, this value may be adjusted.

### 21.5 Known failing tests

If you’re writing a test, you may mark it as a known failing test with the knownfailureif() decorator. This allows the test to be added to the test suite and run on the buildbots without causing undue alarm. For example, although the following test will fail, it is an expected failure:

```python
from nose.tools import assert_equal
from matplotlib.testing.decorators import knownfailureif

@knownfailureif(True)
def test_simple_fail():
    "very simple example test that should fail"
    assert_equal(1+1,3)
```

Note that the first argument to the knownfailureif() decorator is a fail condition, which can be a value such as True, False, or ‘indeterminate’, or may be a dynamically evaluated expression.
21.6 Creating a new module in matplotlib.tests

We try to keep the tests categorized by the primary module they are testing. For example, the tests related to the mathtext.py module are in test_mathtext.py.

Let's say you've added a new module named whizbang.py and you want to add tests for it in matplotlib.tests.test_whizbang. To add this module to the list of default tests, append its name to default_test_modules in lib/matplotlib/__init__.py.

21.7 Using Travis CI

Travis CI is a hosted CI system “in the cloud”.

Travis is configured to receive notifications of new commits to GitHub repos (via GitHub “service hooks”) and to run builds or tests when it sees these new commits. It looks for a YAML file called .travis.yml in the root of the repository to see how to test the project.

Travis CI is already enabled for the main matplotlib GitHub repository – for example, see its Travis page.

If you want to enable Travis CI for your personal matplotlib GitHub repo, simply enable the repo to use Travis CI in either the Travis CI UI or the GitHub UI (Admin | Service Hooks). For details, see the Travis CI Getting Started page. This generally isn’t necessary, since any pull request submitted against the main matplotlib repository will be tested.

Once this is configured, you can see the Travis CI results at http://travis-ci.org/#!your_GitHub_user_name/matplotlib – here’s an example.

21.8 Using tox

Tox is a tool for running tests against multiple Python environments, including multiple versions of Python (e.g., 2.6, 2.7, 3.2, etc.) and even different Python implementations altogether (e.g., CPython, PyPy, Jython, etc.)

Testing all versions of Python (2.6, 2.7, 3.6*) requires having multiple versions of Python installed on your system and on the PATH. Depending on your operating system, you may want to use your package manager (such as apt-get, yum or MacPorts) to do this.

Tox makes it easy to determine if your working copy introduced any regressions before submitting a pull request. Here’s how to use it:

$ pip install tox
$ tox

You can also run tox on a subset of environments:

$ tox -e py26,py27

Tox processes everything serially so it can take a long time to test several environments. To speed it up, you might try using a new, parallelized version of tox called detox. Give this a try:
$ pip install -U -i http://pypi.testrun.org detox
$ detox

Tox is configured using a file called tox.ini. You may need to edit this file if you want to add new environments to test (e.g., py33) or if you want to tweak the dependencies or the way the tests are run. For more info on the tox.ini file, see the Tox Configuration Specification.
22.1 Getting started

The documentation for matplotlib is generated from ReStructured Text using the Sphinx documentation generation tool. Sphinx-1.0 or later and numpydoc 0.4 or later is required.

The documentation sources are found in the doc/ directory in the trunk. To build the users guide in html format, cd into doc/ and do:

```python
python make.py html
```

or:

```bash
./make.py html
```

you can also pass a latex flag to make.py to build a pdf, or pass no arguments to build everything.

The output produced by Sphinx can be configured by editing the conf.py file located in the doc/.

22.2 Organization of matplotlib’s documentation

The actual ReStructured Text files are kept in doc/users, doc/devel, doc/api and doc/faq. The main entry point is doc/index.rst, which pulls in the index.rst file for the users guide, developers guide, api reference, and faqs. The documentation suite is built as a single document in order to make the most effective use of cross referencing, we want to make navigating the Matplotlib documentation as easy as possible.

Additional files can be added to the various guides by including their base file name (the .rst extension is not necessary) in the table of contents. It is also possible to include other documents through the use of an include statement, such as:

```rst
.. include:: ../../TODO
```

22.2.1 docstrings

In addition to the “narrative” documentation described above, matplotlib also defines its API reference documentation in docstrings. For the most part, these are standard Python docstrings, but matplotlib also
Matplotlib, Release 1.4.3

includes some features to better support documenting getters and setters.

Matplotlib uses artist introspection of docstrings to support properties. All properties that you want to
support through setp and getp should have a set_property and get_property method in the Artist
class. Yes, this is not ideal given python properties or enthought traits, but it is a historical legacy for now.
The setter methods use the docstring with the ACCEPTS token to indicate the type of argument the method
accepts. e.g., in matplotlib.lines.Line2D:

```
# in lines.py
def set_linestyle(self, linestyle):
    """
    Set the linestyle of the line
    ACCEPTS: ['-' | '--' | '-.' | ':' | 'steps' | 'None' | '' | '']
    """
```

Since matplotlib uses a lot of pass-through kwargs, e.g., in every function that creates a line (plot(),
semilogx(), semilogy(), etc...), it can be difficult for the new user to know which kwargs are supported.
Matplotlib uses a docstring interpolation scheme to support documentation of every function that takes a
**kwargs. The requirements are:

1. single point of configuration so changes to the properties don’t require multiple docstring edits.

2. as automated as possible so that as properties change, the docs are updated automagically.

The functions matplotlib.artist.kwdocd and matplotlib.artist.kwdoc() to facilitate this. They
combine python string interpolation in the docstring with the matplotlib artist introspection facility that
underlies setp and getp. The kwdocd is a single dictionary that maps class name to a docstring of kwargs.
Here is an example from matplotlib.lines:

```
# in lines.py
artist.kwdocd['Line2D'] = artist.kwdoc(Line2D)
```

Then in any function accepting Line2D pass-through kwargs, e.g., matplotlib.axes.Axes.plot():

```
# in axes.py
def plot(self, *args, **kwargs):
    """
    Some stuff omitted

    The kwargs are Line2D properties:
    %(Line2D)s

    kwargs scalex and scaley, if defined, are passed on
to autoscale_view to determine whether the x and y axes are
autoscaled; default True. See Axes.autoscale_view for more
information
    """
    pass
plot.__doc__ = cbook.dedent(plot.__doc__) % artist.kwdocd
```

Note there is a problem for Artist __init__ methods, e.g., matplotlib.patches.Patch.__init__(),
which supports Patch kwargs, since the artist inspector cannot work until the class is fully defined
and we can’t modify the Patch.__init__.__doc__ docstring outside the class definition. There are
some manual hacks in this case, violating the “single entry point” requirement above – see the `artist.kwdocd['Patch']` setting in `matplotlib.patches`.

### 22.3 Formatting

The Sphinx website contains plenty of documentation concerning ReST markup and working with Sphinx in general. Here are a few additional things to keep in mind:

- Please familiarize yourself with the Sphinx directives for inline markup. Matplotlib’s documentation makes heavy use of cross-referencing and other semantic markup. For example, when referring to external files, use the `:file:` directive.

- Function arguments and keywords should be referred to using the `emphasis` role. This will keep matplotlib’s documentation consistent with Python’s documentation:

  Here is a description of *argument*

  Please do not use the default role:

  Please do not describe `argument` like this.

  nor the literal role:

  Please do not describe `"argument"` like this.

- Sphinx does not support tables with column- or row-spanning cells for latex output. Such tables cannot be used when documenting matplotlib.

- Mathematical expressions can be rendered as png images in html, and in the usual way by latex. For example:

  \[ \sin(x_n^2) \]

  yields: \( \sin(x_n^2) \), and:

  \[
  \int_{-\infty}^{\infty} \frac{e^{i\phi}}{1+x^2 e^{i\phi}} \frac{e^{i\phi}}{1+x^2} \]

  yields:

  \[
  \frac{e^{i\phi}}{1+x^2 e^{i\phi}} \frac{e^{i\phi}}{1+x^2} \]

- Interactive IPython sessions can be illustrated in the documentation using the following directive:

  .. sourcecode:: ipython

  In [69]: lines = plot([1,2,3])

  which would yield:

  In [69]: lines = plot([1,2,3])
Footnotes \(^1\) can be added using \([\#]_\), followed later by:

```
.. rubric:: Footnotes

```

```
.. [\#]
```

Use the `note` and `warning` directives, sparingly, to draw attention to important comments:

```
.. note::
    Here is a note
```

yields:

```
Note: here is a note
```

also:

```
Warning: here is a warning
```

Use the `deprecated` directive when appropriate:

```
.. deprecated:: 0.98
    This feature is obsolete, use something else.
```

yields:

```
Deprecated since version 0.98: This feature is obsolete, use something else.
```

Use the `versionadded` and `versionchanged` directives, which have similar syntax to the `deprecated` role:

```
.. versionadded:: 0.98
    The transforms have been completely revamped.
```

New in version 0.98: The transforms have been completely revamped.

Use the `seealso` directive, for example:

```
.. seealso::

    Using ReST :ref:`emacs-helpers`
    One example

    A bit about :ref:`referring-to-mpl-docs`
    One more
```

yields:

```
See also:

Using ReST `Emacs helpers`: One example

A bit about `Referring to mpl documents`: One more
```

\(^1\) For example.
• Please keep the *Glossary* in mind when writing documentation. You can create a references to a term in the glossary with the `:term:` role.

• The autodoc extension will handle index entries for the API, but additional entries in the index need to be explicitly added.

• Please limit the text width of docstrings to 70 characters.

• Keyword arguments should be described using a definition list.

**Note:** matplotlib makes extensive use of keyword arguments as pass-through arguments, there are many cases where a table is used in place of a definition list for autogenerated sections of docstrings.

## 22.4 Figures

### 22.4.1 Dynamically generated figures

Figures can be automatically generated from scripts and included in the docs. It is not necessary to explicitly save the figure in the script, this will be done automatically at build time to ensure that the code that is included runs and produces the advertised figure.

The path should be relative to the doc directory. Any plots specific to the documentation should be added to the doc/pyplots directory and committed to git. Plots from the examples directory may be referenced through the symlink mpl_examples in the doc directory. e.g.:

```latex
.. plot:: mpl_examples/pylab_examples/simple_plot.py
```

The :scale: directive rescales the image to some percentage of the original size, though we don’t recommend using this in most cases since it is probably better to choose the correct figure size and dpi in mpl and let it handle the scaling.

**Plot directive documentation**

A directive for including a matplotlib plot in a Sphinx document.

By default, in HTML output, `plot` will include a .png file with a link to a high-res .png and .pdf. In LaTeX output, it will include a .pdf.

The source code for the plot may be included in one of three ways:

1. **A path to a source file** as the argument to the directive:

   ```latex
   .. plot:: path/to/plot.py
   ```

   When a path to a source file is given, the content of the directive may optionally contain a caption for the plot:

   ```latex
   .. plot:: path/to/plot.py
   
   This is the caption for the plot
   ```
Additionally, one may specify the name of a function to call (with no arguments) immediately after importing the module:

```
.. plot:: path/to/plot.py plot_function1
```

2. Included as inline content to the directive:

```
.. plot::

    import matplotlib.pyplot as plt
    import matplotlib.image as mpimg
    import numpy as np
    img = mpimg.imread('_static/stinkbug.png')
    imgplot = plt.imshow(img)
```

3. Using doctest syntax:

```
.. plot::

    A plotting example:
    >>> import matplotlib.pyplot as plt
    >>> plt.plot([1,2,3], [4,5,6])
```

**Options**

The `plot` directive supports the following options:

- **format** [{‘python’, ‘doctest’}] Specify the format of the input
- **include-source** [bool] Whether to display the source code. The default can be changed using the `plot_include_source` variable in `conf.py`
- **encoding** [str] If this source file is in a non-UTF8 or non-ASCII encoding, the encoding must be specified using the `:encoding:` option. The encoding will not be inferred using the `-*- coding -*` metacomment.
- **context** [bool or str] If provided, the code will be run in the context of all previous plot directives for which the `:context:` option was specified. This only applies to inline code plot directives, not those run from files. If the `:context: reset` is specified, the context is reset for this and future plots.
- **nofigs** [bool] If specified, the code block will be run, but no figures will be inserted. This is usually useful with the `:context:` option.

Additionally, this directive supports all of the options of the `image` directive, except for `target` (since `plot` will add its own target). These include `alt`, `height`, `width`, `scale`, `align` and `class`.

**Configuration options**

The `plot` directive has the following configuration options:

- **plot_include_source** Default value for the include-source option
- **plot_html_show_source_link** Whether to show a link to the source in HTML.
**plot_pre_code** Code that should be executed before each plot.

**plot_basedir** Base directory, to which `plot::` file names are relative to. (If None or empty, file names are relative to the directory where the file containing the directive is.)

**plot_formats** File formats to generate. List of tuples or strings:

```
[(suffix, dpi), suffix, ...]
```

that determine the file format and the DPI. For entries whose DPI was omitted, sensible defaults are chosen.

**plot_html_show_formats** Whether to show links to the files in HTML.

**plot_rcparams** A dictionary containing any non-standard rcParams that should be applied before each plot.

**plot_apply_rcparams** By default, rcParams are applied when context option is not used in a plot directive. This configuration option overrides this behavior and applies rcParams before each plot.

**plot_working_directory** By default, the working directory will be changed to the directory of the example, so the code can get at its data files, if any. Also its path will be added to `sys.path` so it can import any helper modules sitting beside it. This configuration option can be used to specify a central directory (also added to `sys.path`) where data files and helper modules for all code are located.

**plot_template** Provide a customized template for preparing restructured text.

### 22.4.2 Static figures

Any figures that rely on optional system configurations need to be handled a little differently. These figures are not to be generated during the documentation build, in order to keep the prerequisites to the documentation effort as low as possible. Please run the `doc/pyplots/make.py` script when adding such figures, and commit the script and the images to git. Please also add a line to the README in `doc/pyplots` for any additional requirements necessary to generate a new figure. Once these steps have been taken, these figures can be included in the usual way:

```
.. plot:: pyplots/tex_unicode_demo.py
   :include-source:
```

### 22.4.3 Examples

The source of the files in the `examples` directory are automatically included in the HTML docs. An image is generated and included for all examples in the `api` and `pylab_examples` directories. To exclude the example from having an image rendered, insert the following special comment anywhere in the script:

```
# -*- noplot -*-
```
22.4.4 Animations

We have a matplotlib google/gmail account with username mplgithub which we used to setup the github account but can be used for other purposes, like hosting google docs or youtube videos. You can embed a matplotlib animation in the docs by first saving the animation as a movie using matplotlib.animation.Animation.save(), and then uploading to matplotlib’s youtube channel and inserting the embedding string youtube provides like:

```python
.. raw:: html

    <iframe width="420" height="315"
            src="http://www.youtube.com/embed/32cjc6V0OZY"
            frameborder="0" allowfullscreen>
    </iframe>
```

An example save command to generate a movie looks like this

```python
ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
                              interval=25, blit=True, init_func=init)

ani.save('double_pendulum.mp4', fps=15)
```

Contact Michael Droettboom for the login password to upload youtube videos of google docs to the mplgithub account.

22.5 Referring to mpl documents

In the documentation, you may want to include to a document in the matplotlib src, e.g., a license file or an image file from mpl-data, refer to it via a relative path from the document where the rst file resides, e.g., in users/navigation_toolbar.rst, we refer to the image icons with:

```python
.. image:: ../../lib/matplotlib/mpl-data/images/subplots.png
```

In the users subdirectory, if I want to refer to a file in the mpl-data directory, I use the symlink directory. For example, from customizing.rst:

```python
.. literalinclude:: ../../lib/matplotlib/mpl-data/matplotlibrc
```

One exception to this is when referring to the examples dir. Relative paths are extremely confusing in the sphinx plot extensions, so without getting into the dirty details, it is easier to simply include a symlink to the files at the top doc level directory. This way, API documents like `matplotlib.pyplot.plot()` can refer to the examples in a known location.

In the top level doc directory we have symlinks pointing to the mpl examples:

```bash
home:/mpl/doc> ls -l mpl_
ml_examples -> ../examples
```

So we can include plots from the examples dir using the symlink:
We used to use a symlink for mpl-data too, but the distro becomes very large on platforms that do not support links (e.g., the font files are duplicated and large)

## 22.6 Internal section references

To maximize internal consistency in section labeling and references, use hyphen separated, descriptive labels for section references, e.g.,

```
.. _howto-webapp:
```

and refer to it using the standard reference syntax:

```
See :ref:`howto-webapp`
```

Keep in mind that we may want to reorganize the contents later, so let’s avoid top level names in references like user or devel or faq unless necessary, because for example the FAQ “what is a backend?” could later become part of the users guide, so the label:

```
.. _what-is-a-backend
```

is better than:

```
.. _faq-backend
```

In addition, since underscores are widely used by Sphinx itself, let’s prefer hyphens to separate words.

## 22.7 Section names, etc

For everything but top level chapters, please use Upper lower for section titles, e.g., Possible hangups rather than Possible Hangups

## 22.8 Inheritance diagrams

Class inheritance diagrams can be generated with the inheritance-diagram directive. To use it, you provide the directive with a number of class or module names (separated by whitespace). If a module name is provided, all classes in that module will be used. All of the ancestors of these classes will be included in the inheritance diagram.

A single option is available: parts controls how many of parts in the path to the class are shown. For example, if parts == 1, the class matplotlib.patches.Patch is shown as Patch. If parts == 2, it is shown as patches.Patch. If parts == 0, the full path is shown.

Example:
22.9 Emacs helpers

There is an emacs mode rst.el which automates many important ReST tasks like building and updating table-of-contents, and promoting or demoting section headings. Here is the basic .emacs configuration:

(requires 'rst)
(setq auto-mode-alist
      (append '(
                ("\\.txt$" . rst-mode)
                ("\\.rst$" . rst-mode)
                ("\\.rest$" . rst-mode)) auto-mode-alist))

Some helpful functions:

C-c TAB - rst-toc-insert

Insert table of contents at point
C-c C-u rst-toc-update

Update the table of contents at point

C-c C-l rst-shift-region-left

Shift region to the left

C-c C-r rst-shift-region-right

Shift region to the right
A guide for developers who are doing a matplotlib release.

- Edit `__init__.py` and bump the version number

### 23.1 Testing

- Run all of the regression tests by running the `tests.py` script at the root of the source tree.
- Run `unit/memleak_hawaii3.py` and make sure there are no memory leaks
- Try some GUI examples, e.g., `simple_plot.py` with GTKAgg, TkAgg, etc...
- Remove font cache and tex cache from `.matplotlib` and test with and without cache on some example script
- Optionally, make sure `examples/tests/backend_driver.py` runs without errors and check the output of the PNG, PDF, PS and SVG backends

### 23.2 Branching

Once all the tests are passing and you are ready to do a release, you need to create a release branch. These only need to be created when the second part of the version number changes:

```bash
git checkout -b v1.1.x
git push git@github.com:matplotlib/matplotlib.git v1.1.x
```

On the branch, do any additional testing you want to do, and then build binaries and source distributions for testing as release candidates.

For each release candidate as well as for the final release version, please `git tag` the commit you will use for packaging like so:

```bash
git tag -a v1.1.0rc1
```

The `-a` flag will allow you to write a message about the tag, and affiliate your name with it. A reasonable tag message would be something like `v1.1.0 Release Candidate 1 (September 24, 2011)`. To tag a release after the fact, just track down the commit hash, and:

```bash
```
git tag -a v1.0.1rc1 a9f3f3a50745

Tags allow developers to quickly checkout different releases by name, and also provides source download via zip and tarball on github.
Then push the tags to the main repository:
git push upstream v1.0.1rc1

### 23.3 Packaging

- Make sure the MANIFEST.in is up to date and remove MANIFEST so it will be rebuilt by MANIFEST.in
- run git clean in the mpl git directory before building the sdist
- unpack the sdist and make sure you can build from that directory
- Use setup.cfg to set the default backends. For windows and OSX, the default backend should be TkAgg. You should also turn on or off any platform specific build options you need. Importantly, you also need to make sure that you delete the build dir after any changes to setup.cfg before rebuilding since cruft in the build dir can get carried along.
- On windows, unix2dos the rc file.
- We have a Makefile for the OS X builds in the mpl source dir release/osx, so use this to prepare the OS X releases.
- We have a Makefile for the win32 mingw builds in the mpl source dir release/win32 which you can use this to prepare the windows releases.

### 23.4 Posting files

Our current method is for the release manager to collect all of the binaries from the platform builders and post the files online on Sourceforge. It is also possible that those building the binaries could upload to directly to Sourceforge. We also post a source tarball to PyPI, since pip no longer trusts files downloaded from other sites.

There are many ways to upload files to Sourceforge (scp, rsync, sftp, and a web interface) described in Sourceforge Release File System documentation. Below, we will use sftp.

1. Create a directory containing all of the release files and cd to it.
2. sftp to Sourceforge:

   sftp USERNAME@frs.sourceforge.net:/home/frs/project/matplotlib/matplotlib

3. Make a new directory for the release and move to it:

   mkdir matplotlib-1.1.0rc1
cd matplotlib-1.1.0rc1
4. Upload all of the files in the current directory on your local machine:

    put *

If this release is a final release, the default download for the matplotlib project should also be updated. Login to Sourceforge and visit the matplotlib files page. Navigate to the tarball of the release you just updated, click on “Details” icon (it looks like a lower case i), and make it the default download for all platforms.

There is a list of direct links to downloads on matplotlib’s main website. This needs to be manually generated and updated every time new files are posted.

1. Clone the matplotlib documentation repository and cd into it:

git clone git@github.com:matplotlib/matplotlib.github.com.git
cd matplotlib.github.com

2. Update the list of downloads that you want to display by editing the downloads.txt file. Generally, this should contain the last two final releases and any active release candidates.

3. Update the downloads webpage by running the update_downloads.py script. This script requires paramiko (for sftp support) and jinja2 for templating. Both of these dependencies can be installed using pip:

    pip install paramiko
    pip install jinja2

    Then update the download page:
    ./update_downloads.py

    You will be prompted for your Sourceforge username and password.

4. Commit the changes and push them up to github:

    git commit -m "Updating download list"
    git push

## 23.5 Update PyPI

Once the tarball has been posted on Sourceforge, you can register a link to the new release on PyPI. This should only be done with final (non-release-candidate) releases, since doing so will hide any available stable releases.

You may need to set up your pypirc file as described in the distutils register command documentation.

Then updating the record on PyPI is as simple as:

    python setup.py register

This will hide any previous releases automatically.

Then, to upload the source tarball:
rm -rf dist
python setup.py sdist upload

## 23.6 Documentation updates

The built documentation exists in the matplotlib.github.com repository. Pushing changes to master automatically updates the website.

The documentation is organized by version. At the root of the tree is always the documentation for the latest stable release. Under that, there are directories containing the documentation for older versions as well as the bleeding edge release version called dev (usually based on what’s on master in the github repository, but it may also temporarily be a staging area for proposed changes). There is also a symlink directory with the name of the most recently released version that points to the root. With each new release, these directories may need to be reorganized accordingly. Any time these version directories are added or removed, the versions.html file (which contains a list of the available documentation versions for the user) must also be updated.

To make sure everyone’s hard work gets credited, regenerate the github stats. cd into the tools directory and run:

```
python github_stats.py $TAG > ../doc/users/github_stats.rst
```

where $TAG is the tag of the last major release. This will generate stats for all work done since that release.

In the matplotlib source repository, build the documentation:

```
cd doc
python make.py html
python make.py latex
```

Then copy the build products into your local checkout of the matplotlib.github.com repository (assuming here to be checked out in com):

```
cp -r build/html/* ~/matplotlib.github.com
cp build/latex/Matplotlib.pdf ~/matplotlib.github.com
```

Then, from the matplotlib.github.com directory, commit and push the changes upstream:

```
git commit -m "Updating for v1.0.1"
git push upstream master
```

## 23.7 Announcing

Announce the release on matplotlib-announce, matplotlib-users, and matplotlib-devel. Final (non-release-candidate) versions should also be announced on python-announce. Include a summary of highlights from the CHANGELOG and/or post the whole CHANGELOG since the last release.
24.1 `matplotlib.transforms`

`matplotlib` includes a framework for arbitrary geometric transformations that is used to determine the final position of all elements drawn on the canvas.

Transforms are composed into trees of `TransformNode` objects whose actual value depends on their children. When the contents of children change, their parents are automatically invalidated. The next time an invalidated transform is accessed, it is recomputed to reflect those changes. This invalidation/caching approach prevents unnecessary recomputations of transforms, and contributes to better interactive performance.

For example, here is a graph of the transform tree used to plot data to the graph:
The framework can be used for both affine and non-affine transformations. However, for speed, we want use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform:

\[
\text{full transform} = \text{non-affine part} + \text{affine part}
\]

The backends are not expected to handle non-affine transformations themselves.
class matplotlib.transforms.TransformNode(shorthand_name=None)
Bases: object

TransformNode is the base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

Creates a new TransformNode.

shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

invalidate()
Invalidate this TransformNode and triggers an invalidation of its ancestors. Should be called any time the transform changes.

pass_through = False
If pass_through is True, all ancestors will always be invalidated, even if ‘self’ is already invalid.

set_children(*children)
Set the children of the transform, to let the invalidation system know which transforms can invalidate this transform. Should be called from the constructor of any transforms that depend on other transforms.

class matplotlib.transforms.BboxBase(shorthand_name=None)
Bases: matplotlib.transforms.TransformNode

This is the base class of all bounding boxes, and provides read-only access to its data. A mutable bounding box is provided by the Bbox class.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

Creates a new TransformNode.

shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

anchored(c, container=None)
Return a copy of the Bbox, shifted to position c within a container.

  c: may be either:

  • a sequence (cx, cy) where cx and cy range from 0 to 1, where 0 is left or bottom and 1 is right or top

  • a string: - ‘C’ for centered - ‘S’ for bottom-center - ‘SE’ for bottom-left - ‘E’ for left - etc.

Optional argument container is the box within which the Bbox is positioned; it defaults to the initial Bbox.
bounds
    (property) Returns \((x_0, y_0, \text{width}, \text{height})\).

contains\((x, y)\)
    Returns \(True\) if \((x, y)\) is a coordinate inside the bounding box or on its edge.

containsx\((x)\)
    Returns \(True\) if \(x\) is between or equal to \(x_0\) and \(x_1\).

containsy\((y)\)
    Returns \(True\) if \(y\) is between or equal to \(y_0\) and \(y_1\).

corners()
    Return an array of points which are the four corners of this rectangle. For example, if this Bbox
    is defined by the points \((a, b)\) and \((c, d)\), corners() returns \((a, b), (a, d), (c, b), (c, d)\).

count_contains\((\text{vertices})\)
    Count the number of vertices contained in the Bbox.

    vertices is a Nx2 Numpy array.

count_overlaps\((\text{bboxes})\)
    Count the number of bounding boxes that overlap this one.

    bboxes is a sequence of BboxBase objects

expanded\((\text{sw, sh})\)
    Return a new Bbox which is this Bbox expanded around its center by the given factors \(\text{sw}\) and \(\text{sh}\).

eextents
    (property) Returns \((x_0, y_0, x_1, y_1)\).

frozen()
    TransformNode is the base class for anything that participates in the transform tree and needs to
    invalidate its parents or be invalidated. This includes classes that are not really transforms, such as
    bounding boxes, since some transforms depend on bounding boxes to compute their values.

fully_contains\((x, y)\)
    Returns \(True\) if \((x, y)\) is a coordinate inside the bounding box, but not on its edge.

fully_containsx\((x)\)
    Returns \(True\) if \(x\) is between but not equal to \(x_0\) and \(x_1\).

fully_containssy\((y)\)
    Returns \(True\) if \(y\) is between but not equal to \(y_0\) and \(y_1\).

fully_overlaps\((\text{other})\)
    Returns \(True\) if this bounding box overlaps with the given bounding box \(\text{other}\), but not on its
    edge alone.

height
    (property) The height of the bounding box. It may be negative if \(y_1 < y_0\).
**static intersection** *(bbox1, bbox2)*

Return the intersection of the two bboxes or None if they do not intersect.

Implements the algorithm described at:

http://www.tekpool.com/node/2687

**intervalx**

(property) **intervalx** is the pair of x coordinates that define the bounding box. It is not guaranteed to be sorted from left to right.

**intervaly**

(property) **intervaly** is the pair of y coordinates that define the bounding box. It is not guaranteed to be sorted from bottom to top.

**inverse_transformed**(transform)

Return a new **Bbox** object, statically transformed by the inverse of the given transform.

**is_unit**

Returns True if the **Bbox** is the unit bounding box from (0, 0) to (1, 1).

**max**

(property) **max** is the top-right corner of the bounding box.

**min**

(property) **min** is the bottom-left corner of the bounding box.

**overlaps**(other)

Returns True if this bounding box overlaps with the given bounding box other.

**p0**

(property) **p0** is the first pair of (x, y) coordinates that define the bounding box. It is not guaranteed to be the bottom-left corner. For that, use **min**.

**p1**

(property) **p1** is the second pair of (x, y) coordinates that define the bounding box. It is not guaranteed to be the top-right corner. For that, use **max**.

**padded**(p)

Return a new **Bbox** that is padded on all four sides by the given value.

**rotated**(radians)

Return a new bounding box that bounds a rotated version of this bounding box by the given radians. The new bounding box is still aligned with the axes, of course.

**shrunk**(mx, my)

Return a copy of the **Bbox**, shrunk by the factor mx in the x direction and the factor my in the y direction. The lower left corner of the box remains unchanged. Normally mx and my will be less than 1, but this is not enforced.

**shrunk_to_aspect**(box_aspect, container=None, fig_aspect=1.0)

Return a copy of the **Bbox**, shrunk so that it is as large as it can be while having the desired aspect ratio, box_aspect. If the box coordinates are relative—that is, fractions of a larger box such as a figure—then the physical aspect ratio of that figure is specified with fig_aspect, so that box_aspect can also be given as a ratio of the absolute dimensions, not the relative dimensions.
size
  (property) The width and height of the bounding box. May be negative, in the same way as
  width and height.

splitx(*args)
e.g., bbox.splitx(f1, f2, ...)
  Returns a list of new Bbox objects formed by splitting the original one with vertical lines
  at fractional positions f1, f2, ...

splity(*args)
e.g., bbox.splity(f1, f2, ...)
  Returns a list of new Bbox objects formed by splitting the original one with horizontal lines
  at fractional positions f1, f2, ...

transformed(transform)
  Return a new Bbox object, statically transformed by the given transform.

translated(tx, ty)
  Return a copy of the Bbox, statically translated by tx and ty.

static union(bboxes)
  Return a Bbox that contains all of the given bboxes.

width
  (property) The width of the bounding box. It may be negative if x1 < x0.

x0
  (property) x0 is the first of the pair of x coordinates that define the bounding box. x0 is not
  guaranteed to be less than x1. If you require that, use xmin.

x1
  (property) x1 is the second of the pair of x coordinates that define the bounding box. x1 is not
  guaranteed to be greater than x0. If you require that, use xmax.

xmax
  (property) xmax is the right edge of the bounding box.

xmin
  (property) xmin is the left edge of the bounding box.

y0
  (property) y0 is the first of the pair of y coordinates that define the bounding box. y0 is not
  guaranteed to be less than y1. If you require that, use ymin.

y1
  (property) y1 is the second of the pair of y coordinates that define the bounding box. y1 is not
  guaranteed to be greater than y0. If you require that, use ymax.

ymax
  (property) ymax is the top edge of the bounding box.

ymin
  (property) ymin is the bottom edge of the bounding box.
class matplotlib.transforms.Bbox(points, **kwargs)
    Bases: matplotlib.transforms.BboxBase

A mutable bounding box.

points: a 2x2 numpy array of the form [[x0, y0], [x1, y1]]

If you need to create a Bbox object from another form of data, consider the static methods unit(), from_bounds() and from_extents().

static from_bounds(x0, y0, width, height)
    (staticmethod) Create a new Bbox from x0, y0, width and height.
    width and height may be negative.

static from_extents(*args)
    (staticmethod) Create a new Bbox from left, bottom, right and top.
    The y-axis increases upwards.

get_points()
    Get the points of the bounding box directly as a numpy array of the form: [[x0, y0], [x1, y1]].

ignore(value)
    Set whether the existing bounds of the box should be ignored by subsequent calls to update_from_data() or update_from_data_xy().
    value:
    • When True, subsequent calls to update_from_data() will ignore the existing bounds of the Bbox.
    • When False, subsequent calls to update_from_data() will include the existing bounds of the Bbox.

mutated()
    return whether the bbox has changed since init

mutatedx()
    return whether the x-limits have changed since init

mutatedy()
    return whether the y-limits have changed since init

static null()
    (staticmethod) Create a new null Bbox from (inf, inf) to (-inf, -inf).

set(other)
    Set this bounding box from the “frozen” bounds of another Bbox.

set_points(points)
    Set the points of the bounding box directly from a numpy array of the form: [[x0, y0], [x1, y1]].
    No error checking is performed, as this method is mainly for internal use.

static unit()
    (staticmethod) Create a new unit Bbox from (0, 0) to (1, 1).
**update_from_data**(*x, y, ignore=None*)

Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

*x*: a numpy array of *x*-values

*y*: a numpy array of *y*-values

**ignore:**

- when True, ignore the existing bounds of the Bbox.
- when False, include the existing bounds of the Bbox.
- when None, use the last value passed to *ignore()*.

**update_from_data_xy**(*xy, ignore=None, updatex=True, updatey=True*)

Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

*xy*: a numpy array of 2D points

**ignore:**

- when True, ignore the existing bounds of the Bbox.
- when False, include the existing bounds of the Bbox.
- when None, use the last value passed to *ignore()*.

*updatex*: when True, update the x values

*updatey*: when True, update the y values

**update_from_path**(*path, ignore=None, updatex=True, updatey=True*)

Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

*path*: a Path instance

**ignore:**

- when True, ignore the existing bounds of the Bbox.
- when False, include the existing bounds of the Bbox.
- when None, use the last value passed to *ignore()*.

*updatex*: when True, update the x values

*updatey*: when True, update the y values

**class** matplotlib.transforms.TransformedBbox(*bbox, transform, **kwargs*)

Bases: matplotlib.transforms.BboxBase

A Bbox that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

*bbox*: a child Bbox

*transform*: a 2D Transform
get_points()
Get the points of the bounding box directly as a numpy array of the form: \([x0, y0], [x1, y1]\).

class matplotlib.transforms.Transform(shorthand_name=None)
Bases: matplotlib.transforms.TransformNode

The base class of all TransformNode instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of Affine2D.

Subclasses of this class should override the following members (at minimum):

- `input_dims`
- `output_dims`
- `transform()`
- `is_separable`
- `has_inverse`
- `inverted()` (if `has_inverse` is True)

If the transform needs to do something non-standard with matplotlib.path.Path objects, such as adding curves where there were once line segments, it should override:

- `transform_path()`

Creates a new TransformNode.

**shorthand_name** - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

contains_branch(other)
Return whether the given transform is a sub-tree of this transform.

This routine uses transform equality to identify sub-trees, therefore in many situations it is object id which will be used.

For the case where the given transform represents the whole of this transform, returns True.

contains_branch_seperately(other_transform)
Returns whether the given branch is a sub-tree of this transform on each separate dimension.

A common use for this method is to identify if a transform is a blended transform containing an axes’ data transform. e.g.:

\[
x_{\text{isdata}}, y_{\text{isdata}} = \text{trans}.\text{contains\_branch\_seperately}(\text{ax}.\text{transData})
\]

depth
Returns the number of transforms which have been chained together to form this Transform instance.

**Note:** For the special case of a Composite transform, the maximum depth of the two is returned.
get_affine()
Get the affine part of this transform.

get_matrix()
Get the Affine transformation array for the affine part of this transform.

has_inverse = False
True if this transform has a corresponding inverse transform.

input_dims = None
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

inverted()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

x === self.inverted().transform(self.transform(x))

is_separable = False
True if this transform is separable in the x- and y- dimensions.

output_dims = None
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

transform(values)
Performs the transformation on the given array of values.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_affine(values)
Performs only the affine part of this transformation on the given array of values.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to transform(values).

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_angles(angles, pts, radians=False, pushoff=1e-05)
Performs transformation on a set of angles anchored at specific locations.

The angles must be a column vector (i.e., numpy array).
The *pts* must be a two-column numpy array of x,y positions (angle transforms currently only work in 2D). This array must have the same number of rows as *angles*.

*radians* indicates whether or not input angles are given in radians (True) or degrees (False; the default).

*pushoff* is the distance to move away from *pts* for determining transformed angles (see discussion of method below).

The transformed angles are returned in an array with the same size as *angles*.

The generic version of this method uses a very generic algorithm that transforms *pts*, as well as locations very close to *pts*, to find the angle in the transformed system.

**transform_bbox** (*bbox*)

Transform the given bounding box.

Note, for smarter transforms including caching (a common requirement for matplotlib figures), see `TransformedBbox`.

**transform_non_affine** (*values*)

Performs only the non-affine part of the transformation.

**transform** (*values*) is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x *input_dims*) and returns a numpy array of shape (N x *output_dims*).

Alternatively, accepts a numpy array of length *input_dims* and returns a numpy array of length *output_dims*.

**transform_path** (*path*)

Returns a transformed path.

*path*: a Path instance.

In some cases, this transform may insert curves into the path that began as line segments.

**transform_path_affine** (*path*)

Returns a path, transformed only by the affine part of this transform.

*path*: a Path instance.

**transform_path**(*path*) is equivalent to `transform_path_affine(transform_path_non_affine(values))`.

**transform_path_non_affine** (*path*)

Returns a path, transformed only by the non-affine part of this transform.

*path*: a Path instance.

**transform_path**(*path*) is equivalent to `transform_path_affine(transform_path_non_affine(values))`.

**transform_point** (*point*)

A convenience function that returns the transformed copy of a single point.
The point is given as a sequence of length `input_dims`. The transformed point is returned as a sequence of length `output_dims`.

class matplotlib.transforms.TransformWrapper(child)
Bases: matplotlib.transforms.Transform

A helper class that holds a single child transform and acts equivalently to it.

This is useful if a node of the transform tree must be replaced at run time with a transform of a different type. This class allows that replacement to correctly trigger invalidation.

Note that `TransformWrapper` instances must have the same input and output dimensions during their entire lifetime, so the child transform may only be replaced with another child transform of the same dimensions.

`child`: A class:`Transform` instance. This child may later be replaced with `set()`.

**frozen()**

Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

**set(child)**

Replace the current child of this transform with another one.

The new child must have the same number of input and output dimensions as the current child.

class matplotlib.transforms.AffineBase(*args, **kwargs)
Bases: matplotlib.transforms.Transform

The base class of all affine transformations of any number of dimensions.

**get_affine()**

Get the affine part of this transform.

**transform(values)**

Performs the transformation on the given array of values.

Accepts a numpy array of shape `(N x input_dims)` and returns a numpy array of shape `(N x output_dims)`.

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

**transform_affine(values)**

Performs only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape `(N x input_dims)` and returns a numpy array of shape `(N x output_dims)`.

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.
transform_non_affine(points)
Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_path(path)
Returns a transformed path.

path: a Path instance.

In some cases, this transform may insert curves into the path that began as line segments.

transform_path_affine(path)
Returns a path, transformed only by the affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values))

transform_path_non_affine(path)
Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values))

class matplotlib.transforms.Affine2DBase(*args, **kwargs)
Bases: matplotlib.transforms.AffineBase

The base class of all 2D affine transformations.

2D affine transformations are performed using a 3x3 numpy array:

\[
\begin{bmatrix}
a & c & e \\
b & d & f \\
0 & 0 & 1 \\
\end{bmatrix}
\]

This class provides the read-only interface. For a mutable 2D affine transformation, use Affine2D.

Subclasses of this class will generally only need to override a constructor and get_matrix() that generates a custom 3x3 matrix.

frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

inverted()
Return the corresponding inverse transformation.
The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

\[
x == self.inverted().transform(self.transform(x))
\]

**matrix_from_values** *(a, b, c, d, e, f)*

Create a new transformation matrix as a 3x3 numpy array of the form:

\[
\begin{bmatrix}
a & c & e \\
b & d & f \\
0 & 0 & 1
\end{bmatrix}
\]

**to_values**

Return the values of the matrix as a sequence (a,b,c,d,e,f)

**transform_affine** *(points)*

Performs only the affine part of this transformation on the given array of values.

\[
\text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}.
\]

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to transform(values).

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

**transform_point** *(point)*

A convenience function that returns the transformed copy of a single point.

The point is given as a sequence of length input_dims. The transformed point is returned as a sequence of length output_dims.

**class** matplotlib.transforms.Affine2D*(matrix=None, **kwargs)*

Bases: matplotlib.transforms.Affine2DBase

A mutable 2D affine transformation.

Initialize an Affine transform from a 3x3 numpy float array:

\[
\begin{bmatrix}
a & c & e \\
b & d & f \\
0 & 0 & 1
\end{bmatrix}
\]

If matrix is None, initialize with the identity transform.

**clear**

Reset the underlying matrix to the identity transform.

**from_values** *(a, b, c, d, e, f)*

Create a new Affine2D instance from the given values:

\[
\begin{bmatrix}
a & c & e \\
b & d & f \\
0 & 0 & 1
\end{bmatrix}
\]
get_matrix()

Get the underlying transformation matrix as a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

static identity()

(staticmethod) Return a new Affine2D object that is the identity transform.

Unless this transform will be mutated later on, consider using the faster IdentityTransform class instead.

rotate(theta)

Add a rotation (in radians) to this transform in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

rotate_around(x, y, theta)

Add a rotation (in radians) around the point (x, y) in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

rotate_deg(degrees)

Add a rotation (in degrees) to this transform in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

rotate_deg_around(x, y, degrees)

Add a rotation (in degrees) around the point (x, y) in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

scale(sx, sy=None)

Adds a scale in place.

If sy is None, the same scale is applied in both the x- and y-directions.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

set(other)

Set this transformation from the frozen copy of another Affine2DBase object.

set_matrix(mtx)

Set the underlying transformation matrix from a 3x3 numpy array:
skew\((xShear, yShear)\)

Adds a skew in place.

\(xShear\) and \(yShear\) are the shear angles along the \(x\)- and \(y\)-axes, respectively, in radians.

Returns \(self\), so this method can easily be chained with more calls to \(\text{rotate()}\), \(\text{rotate\_deg()}\), \(\text{translate()}\) and \(\text{scale()}\).

skew\_deg\((xShear, yShear)\)

Adds a skew in place.

\(xShear\) and \(yShear\) are the shear angles along the \(x\)- and \(y\)-axes, respectively, in degrees.

Returns \(self\), so this method can easily be chained with more calls to \(\text{rotate()}\), \(\text{rotate\_deg()}\), \(\text{translate()}\) and \(\text{scale()}\).

translate\((tx, ty)\)

Adds a translation in place.

Returns \(self\), so this method can easily be chained with more calls to \(\text{rotate()}\), \(\text{rotate\_deg()}\), \(\text{translate()}\) and \(\text{scale()}\).

class matplotlib.transforms.IdentityTransform(*args, **kwargs)

Bases: matplotlib.transforms.Affine2DBase

A special class that does on thing, the identity transform, in a fast way.

frozen()

Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where \(\text{copy\_.deepcopy()}\) might normally be used.

get\_affine()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to \(self\) does not cause a corresponding update to its inverted copy.

\[x == self.inverted().transform(self.transform(x))\]

get\_matrix()

Get the Affine transformation array for the affine part of this transform.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to \(self\) does not cause a corresponding update to its inverted copy.

\[x == self.inverted().transform(self.transform(x))\]
**transform(points)**
Performs only the non-affine part of the transformation.

```
transform(values) is always equivalent to transform_affine(transform_non_affine(values)).
```

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

**transform_affine(points)**
Performs only the non-affine part of the transformation.

```
transform(values) is always equivalent to transform_affine(transform_non_affine(values)).
```

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

**transform_non_affine(points)**
Performs only the non-affine part of the transformation.

```
transform(values) is always equivalent to transform_affine(transform_non_affine(values)).
```

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

**transform_path(path)**
Returns a path, transformed only by the non-affine part of this transform.

```
path: a Path instance.
```

```
transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).
```

**transform_path_affine(path)**
Returns a path, transformed only by the non-affine part of this transform.

```
path: a Path instance.
```

```
transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).
```

**transform_path_non_affine(path)**
Returns a path, transformed only by the non-affine part of this transform.
class matplotlib.transforms.BlendedGenericTransform(x_transform, y_transform, **kwargs)

Bases: matplotlib.transforms.Transform

A “blended” transform uses one transform for the x-direction, and another transform for the y-direction.

This “generic” version can handle any given child transform in the x- and y-directions.

Create a new “blended” transform using x_transform to transform the x-axis and y_transform to transform the y-axis.

You will generally not call this constructor directly but use the blended_transform_factory() function instead, which can determine automatically which kind of blended transform to create.

frozen()

Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

generate_affine()

Get the affine part of this transform.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

x == self.inverted().transform(self.transform(x))

transform_non_affine(points)

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

class matplotlib.transforms.BlendedAffine2D(x_transform, y_transform, **kwargs)

Bases: matplotlib.transforms.Affine2DBase

A “blended” transform uses one transform for the x-direction, and another transform for the y-direction.

This version is an optimization for the case where both child transforms are of type Affine2DBase.
Create a new “blended” transform using \(x\_\text{transform}\) to transform the \(x\)-axis and \(y\_\text{transform}\) to transform the \(y\)-axis.

Both \(x\_\text{transform}\) and \(y\_\text{transform}\) must be 2D affine transforms.

You will generally not call this constructor directly but use the \texttt{blended\_transform\_factory()} function instead, which can determine automatically which kind of blended transform to create.

\texttt{get\_matrix()}
Get the Affine transformation array for the affine part of this transform.

\texttt{matplotlib.transforms.blended\_transform\_factory}(x\_\text{transform}, y\_\text{transform})
Create a new “blended” transform using \(x\_\text{transform}\) to transform the \(x\)-axis and \(y\_\text{transform}\) to transform the \(y\)-axis.

A faster version of the blended transform is returned for the case where both child transforms are affine.

\textbf{class} \texttt{matplotlib.transforms.CompositeGenericTransform}(a, b, **kwargs)
\textbf{Bases:} \texttt{matplotlib.transforms.Transform}

A composite transform formed by applying transform \(a\) then transform \(b\).

This “generic” version can handle any two arbitrary transformations.

Create a new composite transform that is the result of applying transform \(a\) then transform \(b\).

You will generally not call this constructor directly but use the \texttt{composite\_transform\_factory()} function instead, which can automatically choose the best kind of composite transform instance to create.

\texttt{frozen()}
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where \texttt{copy.deepcopy()} might normally be used.

\texttt{get\_affine()}
Get the affine part of this transform.

\texttt{inverted()}
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to \texttt{self} does not cause a corresponding update to its inverted copy.

\texttt{x == self.inverted().transform(self.transform(x))}

\texttt{transform\_affine(points)}
Performs only the affine part of this transformation on the given array of values.

\texttt{transform(values)} is always equivalent to \texttt{transform\_affine(transform\_non\_affine(values))}.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to \texttt{transform(values)}.

Accepts a numpy array of shape \((N \times \text{input\_dims})\) and returns a numpy array of shape \((N \times \text{output\_dims})\).
Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

**transform_non_affine(points)**

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

**transform_path_non_affine(path)**

Returns a path, transformed only by the non-affine part of this transform.

`path`: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values))`.

```python
class matplotlib.transforms.CompositeAffine2D(a, b, **kwargs)
Bases: matplotlib.transforms.Affine2DBase
A composite transform formed by applying transform a then transform b.
This version is an optimization that handles the case where both a and b are 2D affines.
Create a new composite transform that is the result of applying transform a then transform b.
Both a and b must be instances of `Affine2DBase`.
You will generally not call this constructor directly but use the `composite_transform_factory()` function instead, which can automatically choose the best kind of composite transform instance to create.

get_matrix()
Get the Affine transformation array for the affine part of this transform.
```

```python
class matplotlib.transforms.BboxTransform(boxin, boxout, **kwargs)
Bases: matplotlib.transforms.Affine2DBase
BboxTransform linearly transforms points from one Bbox to another Bbox.
Create a new BboxTransform that linearly transforms points from boxin to boxout.
```
get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.BboxTransformTo(boxout, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

BboxTransformTo is a transformation that linearly transforms points from the unit bounding box to a given Bbox.

Create a new BboxTransformTo that linearly transforms points from the unit bounding box to boxout.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.BboxTransformFrom(boxin, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

BboxTransformFrom linearly transforms points from a given Bbox to the unit bounding box.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.ScaledTranslation(xt, yt, scale_trans, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

A transformation that translates by xt and yt, after xt and yt have been transformad by the given transform scale_trans.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.TransformedPath(path, transform)
Bases: matplotlib.transforms.TransformNode

A TransformedPath caches a non-affine transformed copy of the Path. This cached copy is automatically updated when the non-affine part of the transform changes.

Note: Paths are considered immutable by this class. Any update to the path’s vertices/codes will not trigger a transform recomputation.

Create a new TransformedPath from the given Path and Transform.

get_fully_transformed_path()

Return a fully-transformed copy of the child path.

get_transformed_path_and_affine()

Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation.

get_transformed_points_and_affine()

Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation. Unlike get_transformed_path_and_affine(), no interpolation will be performed.
Modify the endpoints of a range as needed to avoid singularities.

- **vmin, vmax** the initial endpoints.
- **tiny** threshold for the ratio of the interval to the maximum absolute value of its endpoints. If the interval is smaller than this, it will be expanded. This value should be around 1e-15 or larger; otherwise the interval will be approaching the double precision resolution limit.
- **expander** fractional amount by which vmin and vmax are expanded if the original interval is too small, based on tiny.
- **increasing**: [True | False] If True (default), swap vmin, vmax if vmin > vmax

Returns vmin, vmax, expanded and/or swapped if necessary.

If either input is inf or NaN, or if both inputs are 0, returns -expander, expander.
Matplotlib supports the addition of custom procedures that transform the data before it is displayed.

There is an important distinction between two kinds of transformations. Separable transformations, working on a single dimension, are called “scales”, and non-separable transformations, that handle data in two or more dimensions at a time, are called “projections”.

From the user's perspective, the scale of a plot can be set with `set_xscale()` and `set_yscale()`. Projections can be chosen using the `projection` keyword argument to the `plot()` or `subplot()` functions, e.g.:

```python
plot(x, y, projection="custom")
```

This document is intended for developers and advanced users who need to create new scales and projections for matplotlib. The necessary code for scales and projections can be included anywhere: directly within a plot script, in third-party code, or in the matplotlib source tree itself.

### 25.1 Creating a new scale

Adding a new scale consists of defining a subclass of `matplotlib.scale.ScaleBase`, that includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- A function to limit the range of the axis to acceptable values (`limit_range_for_scale()`). A log scale, for instance, would prevent the range from including values less than or equal to zero.
- Locators (major and minor) that determine where to place ticks in the plot, and optionally, how to adjust the limits of the plot to some “good” values. Unlike `limit_range_for_scale()`, which is always enforced, the range setting here is only used when automatically setting the range of the plot.
- Formatters (major and minor) that specify how the tick labels should be drawn.

Once the class is defined, it must be registered with matplotlib so that the user can select it.

A full-fledged and heavily annotated example is in `examples/api/custom_scale_example.py`. There are also some classes in `matplotlib.scale` that may be used as starting points.
25.2 Creating a new projection

Adding a new projection consists of defining a projection axes which subclasses `matplotlib.axes.Axes` and includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- Transformations for the gridlines, ticks and ticklabels. Custom projections will often need to place these elements in special locations, and matplotlib has a facility to help with doing so.
- Setting up default values (overriding `cla()`), since the defaults for a rectilinear axes may not be appropriate.
- Defining the shape of the axes, for example, an elliptical axes, that will be used to draw the background of the plot and for clipping any data elements.
- Defining custom locators and formatters for the projection. For example, in a geographic projection, it may be more convenient to display the grid in degrees, even if the data is in radians.
- Set up interactive panning and zooming. This is left as an “advanced” feature left to the reader, but there is an example of this for polar plots in `matplotlib.projections.polar`.
- Any additional methods for additional convenience or features.

Once the projection axes is defined, it can be used in one of two ways:

- By defining the class attribute name, the projection axes can be registered with `matplotlib.projections.register_projection()` and subsequently simply invoked by name:
  
  ```python
  plt.axes(projection='my_proj_name')
  ```

- For more complex, parameterisable projections, a generic “projection” object may be defined which includes the method `_as_mpl_axes`. `_as_mpl_axes` should take no arguments and return the projection’s axes subclass and a dictionary of additional arguments to pass to the subclass’ `__init__` method. Subsequently a parameterised projection can be initialised with:
  
  ```python
  plt.axes(projection=MyProjection(param1=param1_value))
  ```

  where `MyProjection` is an object which implements a `_as_mpl_axes` method.

A full-fledged and heavily annotated example is in `examples/api/custom_projection_example.py`. The polar plot functionality in `matplotlib.projections.polar` may also be of interest.
25.3 API documentation

25.3.1 matplotlib.scale

class matplotlib.scale.LinearScale(axis, **kwargs)
   Bases: matplotlib.scale.ScaleBase
   The default linear scale.
   get_transform()
      The transform for linear scaling is just the IdentityTransform.
   set_default_locators_and_formatters(axis)
      Set the locators and formatters to reasonable defaults for linear scaling.

class matplotlib.scale.LogScale(axis, **kwargs)
   Bases: matplotlib.scale.ScaleBase
   A standard logarithmic scale. Care is taken so non-positive values are not plotted.
   For computational efficiency (to push as much as possible to Numpy C code in the common cases),
   this scale provides different transforms depending on the base of the logarithm:
      • base 10 (Log10Transform)
      • base 2 (Log2Transform)
      • base e (NaturalLogTransform)
      • arbitrary base (LogTransform)
   basex/basey: The base of the logarithm
   nonposx/nonposy: ['mask' | 'clip'] non-positive values in x or y can be masked as invalid, or clipped
      to a very small positive number
   subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers.
      For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]
      will place 8 logarithmically spaced minor ticks between each major tick.
   get_transform()
      Return a Transform instance appropriate for the given logarithm base.
   limit_range_for_scale(vmin, vmax, minpos)
     Limit the domain to positive values.
   set_default_locators_and_formatters(axis)
      Set the locators and formatters to specialized versions for log scaling.

class matplotlib.scale.ScaleBase
   Bases: object
   The base class for all scales.
   Scales are separable transformations, working on a single dimension.
Any subclasses will want to override:

- `name`
- `get_transform()`
- `set_default_locators_and_formatters()`

And optionally:

- `limit_range_for_scale()`

`get_transform()`

Return the `Transform` object associated with this scale.

`limit_range_for_scale(vmin, vmax, minpos)`

Returns the range `vmin`, `vmax`, possibly limited to the domain supported by this scale.

- `minpos should be the minimum positive value in the data`. This is used by log scales to determine a minimum value.

`set_default_locators_and_formatters(axis)`

Set the `Locator` and `Formatter` objects on the given axis to match this scale.

```python
class matplotlib.scale.SymmetricalLogScale(axis, **kwargs)
Bases: matplotlib.scale.ScaleBase

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter `linthresh` allows the user to specify the size of this range (-`linthresh`, `linthresh`).

`basex/basey`:

The base of the logarithm

`linthreshx/linthreshy`:

The range (-`x`, `x`) within which the plot is linear (to avoid having the plot go to infinity around zero).

`subsx/subsy`:

Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

`linscalex/linscaley`:

This allows the linear range (-`linthresh` to `linthresh`) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `linscale` == `1.0` (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

`get_transform()`

Return a `SymmetricalLogTransform` instance.

`set_default_locators_and_formatters(axis)`

Set the locators and formatters to specialized versions for symmetrical log scaling.

`matplotlib.scale.get_scale_docs()`

Helper function for generating docstrings related to scales.
matplotlib.scale.register_scale(scale_class)
    Register a new kind of scale.

    *scale_class* must be a subclass of ScaleBase.

matplotlib.scale.scale_factory(scale, axis, **kwargs)
    Return a scale class by name.

    ACCEPTS: [ linear | log | symlog ]

25.3.2 matplotlib.projections

class matplotlib.projections.ProjectionRegistry
    Bases: object
    Manages the set of projections available to the system.

    get_projection_class(name)
        Get a projection class from its `name`.

    get_projection_names()
        Get a list of the names of all projections currently registered.

    register(*projections)
        Register a new set of projection(s).

matplotlib.projections.get_projection_class(projection=None)
    Get a projection class from its name.

    If `projection` is None, a standard rectilinear projection is returned.

matplotlib.projections.get_projection_names()
    Get a list of acceptable projection names.

matplotlib.projections.process_projection_requirements(figure, *args, **kwargs)
    Handle the args/kwargs to for add_axes/add_subplot(gca, returning:

    (axes_proj_class, proj_class_kwargs, proj_stack_key)

    Which can be used for new axes initialization/identification.

    Note: `kwargs` is modified in place.

matplotlib.projections.polar

class matplotlib.projections.polar.InvertedPolarTransform(axis=None, use_rmin=True)
    Bases: matplotlib.transforms.Transform
    The inverse of the polar transform, mapping Cartesian coordinate space $x$ and $y$ back to $theta$ and $r$.

    inverted()
        Return the corresponding inverse transformation.

25.3. API documentation
The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

\[ x == \text{self.inverted().transform(self.transform(x))} \]

**transform_non_affine(xy)**
Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to } \text{transform_affine(transform_non_affine(values))} \]

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

**class matplotlib.projections.polar.PolarAffine(scale_transform, limits)**
Bases: matplotlib.transforms.Affine2DBase

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

limits is the view limit of the data. The only part of its bounds that is used is ymax (for the radius maximum). The theta range is always fixed to (0, 2pi).

**get_matrix()**
Get the Affine transformation array for the affine part of this transform.

**class matplotlib.projections.polar.PolarAxes(**args, **kwargs)**
Bases: matplotlib.axes._axes.Axes

A polar graph projection, where the input dimensions are theta, r.

Theta starts pointing east and goes anti-clockwise.

**class InvertedPolarTransform(axis=None, use_rmin=True)**
Bases: matplotlib.transforms.Transform

The inverse of the polar transform, mapping Cartesian coordinate space x and y back to theta and r.

**inverted()**
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

\[ x == \text{self.inverted().transform(self.transform(x))} \]

**transform_non_affine(xy)**
Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to } \text{transform_affine(transform_non_affine(values))} \]
In non-affine transformations, this is generally equivalent to \texttt{transform(values)}. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x \texttt{input_dims}) and returns a numpy array of shape (N x \texttt{output_dims}).

Alternatively, accepts a numpy array of length \texttt{input_dims} and returns a numpy array of length \texttt{output_dims}.

\begin{verbatim}
class PolarAxes.PolarAffine(scale_transform, limits)
Bases: matplotlib.transforms.Affine2DBase

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

\texttt{limits} is the view limit of the data. The only part of its bounds that is used is ymax (for the radius maximum). The theta range is always fixed to (0, 2pi).

\texttt{get_matrix()}

Get the Affine transformation array for the affine part of this transform.
\end{verbatim}

\begin{verbatim}
class PolarAxes.PolarTransform(axis=None, use_rmin=True)
Bases: matplotlib.transforms.Transform

The base polar transform. This handles projection \texttt{theta} and \texttt{r} into Cartesian coordinate space \texttt{x} and \texttt{y}, but does not perform the ultimate affine transformation into the correct position.

\texttt{inverted()}

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to \texttt{self} does not cause a corresponding update to its inverted copy.

\texttt{x == self.inverted().transform(self.transform(x))}
\end{verbatim}

\begin{verbatim}
transform_non_affine(tr)
Performs only the non-affine part of the transformation.

\texttt{transform(values)} is always equivalent to \texttt{transform_affine(transform_non_affine(values))}.

In non-affine transformations, this is generally equivalent to \texttt{transform(values)}. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x \texttt{input_dims}) and returns a numpy array of shape (N x \texttt{output_dims}).

Alternatively, accepts a numpy array of length \texttt{input_dims} and returns a numpy array of length \texttt{output_dims}.

\texttt{transform_path_non_affine(path)}

Returns a path, transformed only by the non-affine part of this transform.

\texttt{path}: a \texttt{Path} instance.

\texttt{transform_path(path)} is equivalent to \texttt{transform_path_affine(transform_path_non_affine(path))}.
\end{verbatim}

\begin{verbatim}
class PolarAxes.RadialLocator(base)
Bases: matplotlib.ticker.Locator
\end{verbatim}
Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base Locator (which may be different depending on the scale of the r-axis.

**class** PolarAxes.TheraFormatter

Bases: matplotlib.ticker.Formatter

Used to format the theta tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

PolarAxes.can_pan()

Return True if this axes supports the pan/zoom button functionality.

For polar axes, this is slightly misleading. Both panning and zooming are performed by the same button. Panning is performed in azimuth while zooming is done along the radial.

PolarAxes.can_zoom()

Return True if this axes supports the zoom box button functionality.

Polar axes do not support zoom boxes.

PolarAxes.format_coord(theta, r)

Return a format string formatting the coordinate using Unicode characters.

PolarAxes.get_data_ratio()

Return the aspect ratio of the data itself. For a polar plot, this should always be 1.0

PolarAxes.get_rlabel_position()

**Returns** float

The theta position of the radius labels in degrees.

PolarAxes.get_theta_direction()

Get the direction in which theta increases.

-1: Theta increases in the clockwise direction

1: Theta increases in the counterclockwise direction

PolarAxes.get_theta_offset()

Get the offset for the location of 0 in radians.

PolarAxes.set_rgrids(radii, labels=None, angle=None, fmt=None, **kwargs)

Set the radial locations and labels of the r grids.

The labels will appear at radial distances radii at the given angle in degrees.

labels, if not None, is a len(radii) list of strings of the labels to use at each radius.

If labels is None, the built-in formatter will be used.

Return value is a list of tuples (line, label), where line is Line2D instances and the label is Text instances.

kwargs are optional text properties for the labels:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**ACCEPTS:** sequence of floats

Matplotlib.PolarAxes.set_rlabel_position(value)
Updates the theta position of the radius labels.

**Parameters value**: number

The angular position of the radius labels in degrees.

**`PolarAxes.set_theta_direction(direction)`**

Set the direction in which theta increases.

- *clockwise, -1*: Theta increases in the clockwise direction
- *counterclockwise, anticlockwise, 1*: Theta increases in the counterclockwise direction

**`PolarAxes.set_theta_offset(offset)`**

Set the offset for the location of 0 in radians.

**`PolarAxes.set_theta_zero_location(loc)`**

Sets the location of theta's zero. (Calls set_theta_offset with the correct value in radians under the hood.)

May be one of “N”, “NW”, “W”, “SW”, “S”, “SE”, “E”, or “NE”.

**`PolarAxes.set_thetagrids(angles, labels=None, frac=None, fmt=None, **kwargs)`**

Set the angles at which to place the theta grids (these gridlines are equal along the theta dimension). `angles` is in degrees.

`labels`, if not None, is a `len(angles)` list of strings of the labels to use at each angle.

If `labels` is None, the labels will be `fmt % angle`

`frac` is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples `(line, label)`, where `line` is `Line2D` instances and the `label` is `Text` instances.

kwargs are optional text properties for the labels:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>bgcolor</code> or backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>bbox</code></td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>family</code> or fontfamily or fontname or name</td>
<td>[‘serif’</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fontproperties</code> or font_properties</td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
</tbody>
</table>
Table 25.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontalalignment or ha</td>
<td>['center', 'right', 'left']</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left', 'right', 'center']</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPTS: sequence of floats

class matplotlib.projections.polar.PolarTransform(axis=None, use_rmin=True)
    Bases: matplotlib.transforms.Transform

The base polar transform. This handles projection $\theta$ and $r$ into Cartesian coordinate space $x$ and $y$, but does not perform the ultimate affine transformation into the correct position.

inverted()
    Return the corresponding inverse transformation.

    The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

    $x \equiv$ self.inverted().transform(self.transform($x$))

transform_non_affine($tr$)
    Performs only the non-affine part of the transformation.

    transform(values) is always equivalent to transform_affine(transform_non_affine(values)).
In non-affine transformations, this is generally equivalent to \texttt{transform(values)}. In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input\_dims})\) and returns a numpy array of shape \((N \times \text{output\_dims})\).

Alternatively, accepts a numpy array of length \text{input\_dims} and returns a numpy array of length \text{output\_dims}.

\textbf{transform\_path\_non\_affine(path)}

Returns a path, transformed only by the non-affine part of this transform.

\textit{path}: a \texttt{Path} instance.

\texttt{transform\_path(path)} is equivalent to \texttt{transform\_path\_affine(transform\_path\_non\_affine(values))}.

\textbf{class \texttt{matplotlib.projections.polar.RadialLocator}(base)}

\texttt{Bases: \texttt{matplotlib.ticker.Locator}}

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base \texttt{Locator} (which may be different depending on the scale of the \textit{r}-axis.

\textbf{class \texttt{matplotlib.projections.polar.ThetaFormatter}}

\texttt{Bases: \texttt{matplotlib.ticker.Formatter}}

Used to format the \textit{theta} tick labels. Converts the native unit of radians into degrees and adds a degree symbol.
Part V

Matplotlib AxesGrid Toolkit
The matplotlib AxesGrid toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. While the aspect parameter in matplotlib adjust the position of the single axes, AxesGrid toolkit provides a framework to adjust the position of multiple axes according to their aspects.

**Note:** AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of `axes_grid`. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (`axes_grid1` and `axisartist`). While `axes_grid` namespace is maintained for the backward compatibility, use of `axes_grid1` and `axisartist` is recommended.

**Warning:** `axes_grid` and `axisartist` (but not `axes_grid1`) uses a custom Axes class (derived from the mpl's original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use `axes_grid1` to avoid this, or see how things are different in `axes_grid` and `axisartist` (LINK needed)
OVERVIEW OF AXESGRID TOOLKIT

26.1 What is AxesGrid toolkit?

The matplotlib AxesGrid toolkit is a collection of helper classes, mainly to ease displaying (multiple) images in matplotlib.

**Note:** AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of *axes_grid*. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (*axes_gridl* and *axisartist*). While *axes_grid* namespace is maintained for the backward compatibility, use of *axes_gridl* and *axisartist* is recommended.

**Warning:** *axes_grid* and *axisartist* (but not *axes_gridl*) uses a custom Axes class (derived from the mpl’s original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use *axes_gridl* to avoid this, or see how things are different in *axes_grid* and *axisartist* (LINK needed).

AxesGrid toolkit has two namespaces (*axes_gridl* and *axisartist*). *axisartist* contains custom Axes class that is meant to support for curvilinear grids (e.g., the world coordinate system in astronomy). Unlike mpl’s original Axes class which uses Axes.xaxis and Axes.yaxis to draw ticks, ticklines and etc., Axes in axisartist uses special artist (AxisArtist) which can handle tick, ticklines and etc. for curved coordinate systems.
Since it uses a special artists, some mpl commands that work on Axes.xaxis and Axes.yaxis may not work. See LINK for more detail.

axes_grid1 is a collection of helper classes to ease displaying (multiple) images with matplotlib. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates, which may not be ideal for displaying images that needs to have a given aspect ratio. For example, it helps you to have a colorbar whose height always matches that of the image. ImageGrid, RGB Axes and AxesDivider are helper classes that deals with adjusting the location of (multiple) Axes. They provides a framework to adjust the position of multiple axes at the drawing time. ParasiteAxes provides twinx(or twiny)-like features so that you can plot different data (e.g., different y-scale) in a same Axes. AnchoredArtists includes custom artists which are placed at some anchored position, like the legend.
26.2 AXES_GRID1

26.2.1 ImageGrid

A class that creates a grid of Axes. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio. For example, displaying images of a same size with some fixed padding between them cannot be easily done in matplotlib. ImageGrid is used in such case.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
nrows_ncols = (2, 2),  # creates 2x2 grid of axes
axes_pad=0.1,  # pad between axes in inch.
)

for i in range(4):
    grid[i].imshow(im)  # The AxesGrid object work as a list of axes.

plt.show()
```

- The position of each axes is determined at the drawing time (see AxesDivider), so that the size of the
entire grid fits in the given rectangle (like the aspect of axes). Note that in this example, the paddings between axes are fixed even if you changes the figure size.

- axes in the same column has a same axes width (in figure coordinate), and similarly, axes in the same row has a same height. The widths (height) of the axes in the same row (column) are scaled according to their view limits (xlim or ylim).

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

F = plt.figure(1, (5.5, 3.5))
g = ImageGrid(F, 111,  # similar to subplot(111)
              nrows_ncols = (1, 3),
              axes_pad = 0.1,
              add_all=True,
              label_mode = "L",
             )

Z, extent = get_demo_image()  # demo image
im1=Z
im2=Z[:,10:]
im3=Z[:,10:]
vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = g[i]
    ax.imshow(im, origin="lower", vmin=vmin, vmax=vmax, interpolation="nearest")

plt.draw()
plt.show()
```
- xaxis are shared among axes in a same column. Similarly, yaxis are shared among axes in a same row. Therefore, changing axis properties (view limits, tick location, etc. either by plot commands or using your mouse in interactive backends) of one axes will affect all other shared axes.

When initialized, ImageGrid creates given number (ngrids or ncols * nrows if ngrids is None) of Axes instances. A sequence-like interface is provided to access the individual Axes instances (e.g., grid[0] is the first Axes in the grid. See below for the order of axes).

AxesGrid takes following arguments,

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figrect</td>
<td></td>
<td>number of rows and cols. e.g., (2,2)</td>
</tr>
<tr>
<td>ncols</td>
<td></td>
<td>number of grids. nrows x ncols if None</td>
</tr>
<tr>
<td>ngrids</td>
<td>None</td>
<td>increasing direction of axes number. [row</td>
</tr>
<tr>
<td>direction</td>
<td>&quot;row&quot;</td>
<td>pad between axes in inches</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>Add axes to figures if True</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>xaxis &amp; yaxis of all axes are shared if True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>aspect of axes</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>location of tick labels thaw will be displayed. “1” (only the lower left axes), “L” (left most and bottom most axes), or “all”.</td>
</tr>
<tr>
<td>label_mode</td>
<td>&quot;L&quot;</td>
<td>[None</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>[right</td>
</tr>
<tr>
<td>cbar_location</td>
<td>right</td>
<td>pad between image axes and colorbar axes</td>
</tr>
<tr>
<td>cbar_pad</td>
<td>None</td>
<td>size of the colorbar</td>
</tr>
<tr>
<td>cbar_size</td>
<td>&quot;5%&quot;</td>
<td></td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
rect specifies the location of the grid. You can either specify coordinates of the rectangle to be used (e.g., (0.1, 0.1, 0.8, 0.8) as in the Axes), or the subplot-like position (e.g., “121”).

direction means the increasing direction of the axes number.

aspect By default (False), widths and heights of axes in the grid are scaled independently. If True, they are scaled according to their data limits (similar to aspect parameter in mpl).

share_all if True, xaxis and yaxis of all axes are shared.

direction direction of increasing axes number. For “row”,

\[
\begin{array}{c|c|c|c}
  \text{grid[0]} & \text{grid[1]} \\
  \text{grid[2]} & \text{grid[3]}
\end{array}
\]

For “column”,

\[
\begin{array}{c|c|c|c}
  \text{grid[0]} & \text{grid[2]} \\
  \text{grid[1]} & \text{grid[3]}
\end{array}
\]

You can also create a colorbar (or colorbars). You can have colorbar for each axes (cbar_mode=”each”), or you can have a single colorbar for the grid (cbar_mode=”single”). The colorbar can be placed on your right, or top. The axes for each colorbar is stored as a cbar_axes attribute.

The examples below show what you can do with AxesGrid.

26.2.2 AxesDivider

Behind the scene, the ImageGrid class and the RGBAxes class utilize the AxesDivider class, whose role is to calculate the location of the axes at drawing time. While a more about the AxesDivider is (will be) explained in (yet to be written) AxesDividerGuide, direct use of the AxesDivider class will not be necessary for most users. The axes_divider module provides a helper function make_axes_locatable, which can be useful. It takes a existing axes instance and create a divider for it.

```python
ax = subplot(1,1,1)
divider = make_axes_locatable(ax)
```

**make_axes_locatable** returns an instance of the AxesLocator class, derived from the Locator. It provides **append_axes** method that creates a new axes on the given side of (“top”, “right”, “bottom” and “left”) of the original axes.
26.2.3 colorbar whose height (or width) in sync with the master axes

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import numpy as np

ax = plt.subplot(111)
im = ax.imshow(np.arange(100).reshape((10,10)))
# create an axes on the right side of ax. The width of cax will be 5%
# of ax and the padding between cax and ax will be fixed at 0.05 inch.
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(im, cax=cax)
```

scatter_hist.py with AxesDivider

The “scatter_hist.py” example in mpl can be rewritten using `make_axes_locatable`.

```python
axScatter = subplot(111)
axScatter.scatter(x, y)
axScatter.set_aspect(1.)
```
# create new axes on the right and on the top of the current axes.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", size=1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", size=1.2, pad=0.1, sharey=axScatter)

# the scatter plot:
# histograms
bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

See the full source code below.

The scatter_hist using the AxesDivider has some advantage over the original scatter_hist.py in mpl. For example, you can set the aspect ratio of the scatter plot, even with the x-axis or y-axis is shared accordingly.
26.2.4 ParasiteAxes

The ParasiteAxes is an axes whose location is identical to its host axes. The location is adjusted in the
drawing time, thus it works even if the host change its location (e.g., images).

In most cases, you first create a host axes, which provides a few method that can be used to create parasite
axes. They are `twinx`, `twiny` (which are similar to twinx and twiny in the matplotlib) and `twin`. `twin` takes
an arbitrary transformation that maps between the data coordinates of the host axes and the parasite axes.
`draw` method of the parasite axes are never called. Instead, host axes collects artists in parasite axes and
draw them as if they belong to the host axes, i.e., artists in parasite axes are merged to those of the host axes
and then drawn according to their zorder. The host and parasite axes modifies some of the axes behavior.
For example, color cycle for plot lines are shared between host and parasites. Also, the legend command in
host, creates a legend that includes lines in the parasite axes. To create a host axes, you may use `host_subplot`
or `host_axes` command.

Example 1. twinx

```python
from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt

host = host_subplot(111)
par = host.twinx()

host.set_xlabel("Distance")
host.set_ylabel("Density")
par.set_ylabel("Temperature")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par.plot([0, 1, 2], [0, 3, 2], label="Temperature")

leg = plt.legend()

host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())

par.yaxis.get_label().set_color(p2.get_color())
leg.texts[1].set_color(p2.get_color())

plt.show()
```
Example 2. twin

`twin` without a transform argument treat the parasite axes to have a same data transform as the host. This can be useful when you want the top(or right)-axis to have different tick-locations, tick-labels, or tick-formatter for bottom(or left)-axis.

```python
ax2 = ax.twin()  # now, ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
ax2.set_xticklabels(["0", r"\frac{1}{2}\pi", r"\pi", r"\frac{3}{2}\pi", r"2\pi"])
```
A more sophisticated example using twin. Note that if you change the x-limit in the host axes, the x-limit of the parasite axes will change accordingly.
26.2.5 AnchoredArtists

It's a collection of artists whose location is anchored to the (axes) bbox, like the legend. It is derived from `OffsetBox` in mpl, and artist need to be drawn in the canvas coordinate. But, there is a limited support for an arbitrary transform. For example, the ellipse in the example below will have width and height in the data coordinate.

```python
import matplotlib.pyplot as plt

def draw_text(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredText
    at = AnchoredText("Figure 1a", loc=2, prop=dict(size=8), frameon=True,
                    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)", loc=3, prop=dict(size=8), frameon=True,
                      bbox_to_anchor=(0., 1.),
                      bbox_transform=ax.transAxes
                    )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
```

---

**Linear velocity at 2.3 kpc [km/s]**

**Proper Motion \([\phi]/yr\)**

**FWHM [km/s]**
ax.add_artist(at2)

def draw_circle(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                                loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)

def draw_ellipse(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                         loc=3, pad=0.5, borderpad=0.4, frameon=True)

    ax.add_artist(ae)

def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1\\prime",
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)

    ax.add_artist(asb)

if 1:
    ax = plt.gca()
    ax.set_aspect(1.)

draw_text(ax)
draw_circle(ax)
draw_ellipse(ax)
draw_sizebar(ax)

plt.show()
26.2.6 InsetLocator

`mpl_toolkits.axes_grid.inset_locator` provides helper classes and functions to place your (inset) axes at the anchored position of the parent axes, similarly to AnchoredArtist.

Using `mpl_toolkits.axes_grid.inset_locator.inset_axes()`, you can have inset axes whose size is either fixed, or a fixed proportion of the parent axes. For example:

```python
inset_axes = inset_axes(parent_axes,
                        width="30%",  # width = 30% of parent_bbox
                        height=1.,    # height : 1 inch
                        loc=3)
```

creates an inset axes whose width is 30% of the parent axes and whose height is fixed at 1 inch.

You may creates your inset whose size is determined so that the data scale of the inset axes to be that of the parent axes multiplied by some factor. For example,

```python
inset_axes = zoomed_inset_axes(ax,
                               0.5,  # zoom = 0.5
                               loc=1)
```

creates an inset axes whose data scale is half of the parent axes. Here is complete examples.
For example, `zoomed_inset_axes()` can be used when you want the inset represents the zoom-up of the small portion in the parent axes. And `mpl_toolkits/axes_grid/inset_locator` provides a helper function `mark_inset()` to mark the location of the area represented by the inset axes.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
import numpy as np

def get_demo_image:
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

fig, ax = plt.subplots(figsize=[5,4])

# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30+ny, 30:30+nx] = Z

# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest",
          origin="lower")

axins = zoomed_inset_axes(ax, 6, loc=1) # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest",
             origin="lower")
```
# sub region of the original image
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)
plt.xticks(visible=False)
plt.yticks(visible=False)

# draw a bbox of the region of the inset axes in the parent axes and
# connecting lines between the bbox and the inset axes area
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
plt.draw()
plt.show()

RGB Axes

RGBAxes is a helper class to conveniently show RGB composite images. Like ImageGrid, the location of axes are adjusted so that the area occupied by them fits in a given rectangle. Also, the xaxis and yaxis of each axes are shared.

from mpl_toolkits.axes_grid1.axes_rgb import RGBAxes

fig = plt.figure(1)
ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8])
r, g, b = get_rgb()  # r,g,b are 2-d images
import mpl_toolkits.axisartist as AA
fig = plt.figure(1)
ax = AA.Axes(fig, [0.1, 0.1, 0.8, 0.8])
fig.add_axes(ax)

26.3 AXISARTIST

26.3.1 AxisArtist

AxisArtist module provides a custom (and very experimental) Axes class, where each axis (left, right, top and bottom) have a separate artist associated which is responsible to draw axis-line, ticks, ticklabels, label. Also, you can create your own axis, which can pass through a fixed position in the axes coordinate, or a fixed position in the data coordinate (i.e., the axis floats around when viewlimit changes).

The axes class, by default, have its xaxis and yaxis invisible, and has 4 additional artists which are responsible to draw axis in “left”, “right”, “bottom” and “top”. They are accessed as ax.axis["left"], ax.axis["right"], and so on, i.e., ax.axis is a dictionary that contains artists (note that ax.axis is still a callable methods and it behaves as an original Axes.axis method in mpl).

To create an axes,

```
ax.imshow_rgb(r, g, b,
    origin="lower", interpolation="nearest")
```
or to create a subplot

```python
ax = AA.Subplot(fig, 111)
fig.add_subplot(ax)
```

For example, you can hide the right, and top axis by

```python
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)
```

It is also possible to add an extra axis. For example, you may have an horizontal axis at y=0 (in data coordinate).

```python
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```

```python
import matplotlib.pyplot as plt
import mpl_toolkits.axisartist as AA

fig = plt.figure(1)
fig.subplots_adjust(right=0.85)
ax = AA.Subplot(fig, 1, 1, 1)
fig.add_subplot(ax)

# make some axis invisible
ax.axis["bottom", "top", "right"].set_visible(False)

# make an new axis along the first axis axis (x-axis) which pass 
# throught y=0.
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0,
axis_direction="bottom")

ax.axis["y=0"].toggle(all=True)
ax.axis["y=0"].label.set_text("y = 0")
```
ax.set_ylim(-2, 4)
plt.show()

Or a fixed axis with some offset

    # make new (right-side) yaxis, but with some offset
    ax.axis['right2'] = ax.new_fixed_axis(loc='right',
                                          offset=(20, 0))

**AxisArtist with ParasiteAxes**

Most commands in the axes_grid1 toolkit can take an axes_class keyword argument, and the commands create an axes of the given class. For example, to create a host subplot with axisartist.Axes,

    import mpl_toolkits.axisartist as AA
    from mpl_toolkits.axes_grid1 import host_subplot

    host = host_subplot(111, axes_class=AA.Axes)

Here is an example that uses parasiteAxes.
26.3.2 Curvilinear Grid

The motivation behind the AxisArtist module is to support curvilinear grid and ticks.
See AXISARTIST namespace for more details.

26.3.3 Floating Axes

This also support a Floating Axes whose outer axis are defined as floating axis.
Chapter 26. Overview of AxesGrid toolkit
27.1 AxesDivider

The axes_divider module provide helper classes to adjust the axes positions of set of images in the drawing time.

- **axes_size** provides a classes of units that the size of each axes will be determined. For example, you can specify a fixed size

- **Divider** this is the class that is used calculates the axes position. It divides the given rectangular area into several areas. You initialize the divider by setting the horizontal and vertical list of sizes that the division will be based on. You then use the new_locator method, whose return value is a callable object that can be used to set the axes_locator of the axes.

You first initialize the divider by specifying its grids, i.e., horizontal and vertical.

for example,:  
rect = [0.2, 0.2, 0.6, 0.6]  
horiz=[h0, h1, h2, h3]  
vert=[v0, v1, v2]  
divider = Divider(fig, rect, horiz, vert)

where, rect is a bounds of the box that will be divided and h0...h3, v0...v2 need to be an instance of classes in the axes_size. They have get_size method that returns a tuple of two floats. The first float is the relative size, and the second float is the absolute size. Consider a following grid.

<table>
<thead>
<tr>
<th>v0</th>
<th>v1</th>
<th>h0,v2</th>
<th>h1</th>
<th>h2</th>
<th>h3</th>
</tr>
</thead>
</table>

- v0 => 0, 2
- v1 => 2, 0
- v2 => 3, 0
The height of the bottom row is always 2 (axes_divider internally assumes that the unit is inch). The first and the second rows with height ratio of 2:3. For example, if the total height of the grid 6, then the first and second row will each occupy \( \frac{2}{2+3} \) and \( \frac{3}{2+3} \) of (6-1) inches. The widths of columns (horiz) will be similarly determined. When aspect ratio is set, the total height (or width) will be adjusted accordingly.

The `mpl_toolkits.axes_grid.axes_size` contains several classes that can be used to set the horizontal and vertical configurations. For example, for the vertical configuration above will be:

```python
from mpl_toolkits.axes_grid.axes_size import Fixed, Scaled
vert = [Fixed(2), Scaled(2), Scaled(3)]
```

After you set up the divider object, then you create a locator instance which will be given to the axes:

```python
locator = divider.new_locator(nx=0, ny=1)
ax.set_axes_locator(locator)
```

The return value of the `new_locator` method is a instance of the AxesLocator class. It is a callable object that returns the location and size of the cell at the first column and the second row. You may create a locator that spans over multiple cells:

```python
locator = divider.new_locator(nx=0, nx=2, ny=1)
```

The above locator, when called, will return the position and size of the cells spanning the first and second column and the first row. You may consider it as [0:2, 1].

See the example,

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4.))
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d")%i) for i in range(4)]

horiz = [Size.Scaled(1.5), Size.Fixed(.5), Size.Scaled(1.), Size.Scaled(.5)]
vert = [Size.Scaled(1.), Size.Fixed(.5), Size.Scaled(1.5)]

divider = Divider(fig1, rect, horiz, vert, aspect=False)

for ax1 in ax:
    ax1.set_axes_locator(divider.new_locator(nx=0, ny=0))
    ax1.set_axes_locator(divider.new_locator(nx=0, ny=2))
    ax1.set_axes_locator(divider.new_locator(nx=2, ny=2))
    ax1.set_axes_locator(divider.new_locator(nx=2, nx1=4, ny=0))

    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(), visible=False)
```

504 Chapter 27. The Matplotlib AxesGrid Toolkit User’s Guide
You can adjust the size of the each axes according to their x or y data limits (AxesX and AxesY), similar to the axes aspect parameter.

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4))
# the rect parameter will be ignore as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d") for i in range(4)]

horiz = [Size.AxesX(ax[0]), Size.Fixed(.5), Size.AxesX(ax[1])]
vert = [Size.AxesY(ax[0]), Size.Fixed(.5), Size.AxesY(ax[2])]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)

ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=2, ny=0))
ax[2].set_axes_locator(divider.new_locator(nx=0, ny=2))
```
ax[3].set_axes_locator(divider.new_locator(nx=2, ny=2))

ax[0].set_xlim(0, 2)
ax[1].set_xlim(0, 1)

ax[0].set_ylim(0, 1)
ax[2].set_ylim(0, 2)

divider.set_aspect(1.)

for ax1 in ax:
    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(),
             visible=False)

plt.draw()
plt.show()

27.2 AXISARTIST namespace

The AxisArtist namespace includes a derived Axes implementation. The biggest difference is that the artists responsible to draw axis line, ticks, ticklabel and axis labels are separated out from the mpl’s Axis class, which are much more than artists in the original mpl. This change was strongly motivated to support curvilinear grid. Here are a few things that mpl_toolkits.axisartist.Axes is different from original Axes from mpl.

• Axis elements (axis line(spine), ticks, ticklabel and axis labels) are drawn by a AxisArtist instance.
Unlike Axis, left, right, top and bottom axis are drawn by separate artists. And each of them may have different tick location and different tick labels.

- gridlines are drawn by a Gridlines instance. The change was motivated that in curvilinear coordinate, a gridline may not cross axis-lines (i.e., no associated ticks). In the original Axes class, gridlines are tied to ticks.
- ticklines can be rotated if necessary (i.e, along the gridlines)

In summary, all these changes was to support

- a curvilinear grid.
- a floating axis

mpl_toolkits.axisartist.Axes class defines a axis attribute, which is a dictionary of AxisArtist instances. By default, the dictionary has 4 AxisArtist instances, responsible for drawing of left, right, bottom and top axis. xaxis and yaxis attributes are still available, however they are set to not visible. As separate artists are used for rendering axis, some axis-related method in mpl may have no effect. In addition to AxisArtist instances, the mpl_toolkits.axisartist.Axes will have gridlines attribute (Gridlines), which obviously draws grid lines.

In both AxisArtist and Gridlines, the calculation of tick and grid location is delegated to an instance of GridHelper class. mpl_toolkits.axisartist.Axes class uses GridHelperRectlinear as a grid helper. The Grid-
HelperRectlinear class is a wrapper around the \textit{xaxis} and \textit{yaxis} of mpl’s original Axes, and it was meant to work as the way how mpl’s original axes works. For example, tick location changes using \texttt{set_ticks} method and etc. should work as expected. But change in artist properties (e.g., color) will not work in general, although some effort has been made so that some often-change attributes (color, etc.) are respected.

\section*{27.2.1 \texttt{AxisArtist}}

\texttt{AxisArtist} can be considered as a container artist with following attributes which will draw ticks, labels, etc.

- \texttt{line}
- \texttt{major_ticks, major_ticklabels}
- \texttt{minor_ticks, minor_ticklabels}
- \texttt{offsetText}
- \texttt{label}

\texttt{line}

Derived from \texttt{Line2d} class. Responsible for drawing a spinal(?) line.

\texttt{major_ticks, minor_ticks}

Derived from \texttt{Line2d} class. Note that ticks are markers.

\texttt{major_ticklabels, minor_ticklabels}

Derived from \texttt{Text}. Note that it is not a list of \texttt{Text} artist, but a single artist (similar to a collection).

\texttt{axislabel}

Derived from \texttt{Text}.

\textbf{Default \texttt{AxisArtists}}

By default, following for axis artists are defined:

\begin{verbatim}
ax.axis["left"], ax.axis["bottom"], ax.axis["right"], ax.axis["top"]
\end{verbatim}

The ticklabels and axislabel of the top and the right axis are set to not visible.

For example, if you want to change the color attributes of major_ticklabels of the bottom x-axis

\begin{verbatim}
ax.axis["bottom"].major_ticklabels.set_color("b")
\end{verbatim}

Similarly, to make ticklabels invisible
ax.axis["bottom"].major_ticklabels.set_visible(False)

AxisArtist provides a helper method to control the visibility of ticks, ticklabels, and label. To make ticklabel invisible,

ax.axis["bottom"].toggle(ticklabels=False)

To make all of ticks, ticklabels, and (axis) label invisible

ax.axis["bottom"].toggle(all=False)

To turn all off but ticks on

ax.axis["bottom"].toggle(all=False, ticks=True)

To turn all on but (axis) label off

ax.axis["bottom"].toggle(all=True, label=False)

ax.axis's __getitem__ method can take multiple axis names. For example, to turn ticklabels of "top" and "right" axis on,

ax.axis["top","right"].toggle(ticklabels=True)

Note that ‘ax.axis["top","right"]’ returns a simple proxy object that translate above code to something like below.

for n in ["top","right"]:
    ax.axis[n].toggle(ticklabels=True)

So, any return values in the for loop are ignored. And you should not use it anything more than a simple method.

Like the list indexing ":" means all items, i.e.,

ax.axis[ :].major_ticks.set_color("r")

changes tick color in all axis.

27.2.2 HowTo

1. Changing tick locations and label.
   
   Same as the original mpl’s axes:
   
   ax.set_xticks([1,2,3])

2. Changing axis properties like color, etc.

   Change the properties of appropriate artists. For example, to change the color of the ticklabels:
   
   ax.axis["left"].major_ticklabels.set_color("r")

3. To change the attributes of multiple axis:
ax.axis["left","bottom"].major_ticklabels.set_color("r")

or to change the attributes of all axis:
ax.axis[:].major_ticklabels.set_color("r")

4. **To change the tick size (length), you need to use** `axis.major_ticks.set_ticksize` method. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use `axis.major_ticks.set_tick_out` method.

   To change the pad between ticks and ticklabels, use `axis.major_ticklabels.set_pad` method.

   To change the pad between ticklabels and axis label, `axis.label.set_pad` method.

### 27.2.3 Rotation and Alignment of TickLabels

This is also quite different from the original mpl and can be confusing. When you want to rotate the ticklabels, first consider using “set_axis_direction” method.

```python
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["right"].label.set_axis_direction("left")
```

The parameter for `set_axis_direction` is one of ["left", “right”, “bottom”, “top”].

You must understand some underlying concept of directions.

1. There is a reference direction which is defined as the direction of the axis line with increasing coordinate. For example, the reference direction of the left x-axis is from bottom to top.
The direction, text angle, and alignments of the ticks, ticklabels and axis-label is determined with respect to the reference direction

2. *ticklabel_direction* is either the right-hand side (+) of the reference direction or the lefthand side (-).

3. same for the *label_direction*

4. ticks are by default drawn toward the opposite direction of the ticklabels.
5. text rotation of ticklabels and label is determined in reference to the `ticklabel_direction` or `label_direction`, respectively. The rotation of ticklabels and label is anchored.

On the other hand, there is a concept of “axis_direction”. This is a default setting of above properties for each, “bottom”, “left”, “top”, and “right” axis.

<table>
<thead>
<tr>
<th>?</th>
<th>?</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axislabel</td>
<td>direction</td>
<td>'---'</td>
<td>'---'</td>
<td>'---'</td>
<td>'---'</td>
</tr>
<tr>
<td>axislabel</td>
<td>rotation</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel</td>
<td>va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel</td>
<td>ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>ticklabel</td>
<td>direction</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel</td>
<td>rotation</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>ticklabel</td>
<td>ha</td>
<td>va</td>
<td>baseline</td>
<td>baseline</td>
<td>baseline</td>
</tr>
</tbody>
</table>

And, ‘set_axis_direction(“top”)’ means to adjust the text rotation etc, for settings suitable for “top” axis. The concept of axis direction can be more clear with curved axis.

```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set_axis_direction('top')

plt.show()
```
The axis_direction can be adjusted in the AxisArtist level, or in the level of its child artists, i.e., ticks, ticklabels, and axis-label.

```python
ax1.axis["left"].set_axis_direction("top")
```
changes axis_direction of all the associated artist with the “left” axis, while

```python
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
```
changes the axis_direction of only the major_ticklabels. Note that set_axis_direction in the AxisArtist level changes the ticklabel_direction and label_direction, while changing the axis_direction of ticks, ticklabels, and axis-label does not affect them.

If you want to make ticks outward and ticklabels inside the axes, use invert_ticklabel_direction method.

```python
ax.axis[:].invert_ticklabel_direction()
```
A related method is “set_tick_out”. It makes ticks outward (as a matter of fact, it makes ticks toward the opposite direction of the default direction).

```python
ax.axis[:].major_ticks.set_tick_out(True)
```

So, in summary,

- **AxisArtist’s methods**
  - set_axis_direction : “left”, “right”, “bottom”, or “top”
  - set_ticklabel_direction : “+” or “-”
  - set_axislabel_direction : “+” or “-”
  - invert_ticklabel_direction

- **Ticks’ methods (major_ticks and minor_ticks)**
  - set_tick_out : True or False
  - set_ticksize : size in points

- **TickLabels’ methods (major_ticklabels and minor_ticklabels)**
  - set_axis_direction : “left”, “right”, “bottom”, or “top”
  - set_rotation : angle with respect to the reference direction
- set_ha and set_va: see below

- **AxisLabels’ methods (label)**
  - set_axis_direction: “left”, “right”, “bottom”, or “top”
  - set_rotation: angle with respect to the reference direction
  - set_ha and set_va

**Adjusting ticklabels alignment**

Alignment of TickLabels are treated specially. See below

![Diagram showing alignment options](image)

**Adjusting pad**

To change the pad between ticks and ticklabels

```python
ax.axis['left'].major_ticklabels.set_pad(10)
```

Or ticklabels and axis-label
Matplotlib, Release 1.4.3

ax.axis["left"].label.set_pad(10)

```
default
ticklabels.set_pad(10)  label.set_pad(20)  ticks.set_tick_out(True)
```

27.2.4 GridHelper

To actually define a curvilinear coordinate, you have to use your own grid helper. A generalised version of grid helper class is supplied and this class should suffice in most of cases. A user may provide two functions which defines a transformation (and its inverse pair) from the curved coordinate to (rectilinear) image coordinate. Note that while ticks and grids are drawn for curved coordinate, the data transform of the axes itself (ax.transData) is still rectilinear (image) coordinate.

```
from mpl_toolkits.axisartist.grid_helper_curvelinear import GridHelperCurveLinear
from mpl_toolkits.axisartist import Subplot

# from curved coordinate to rectlinear coordinate.
def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

# from rectlinear coordinate to curved coordinate.
def inv_tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y+x

grid_helper = GridHelperCurveLinear((tr, inv_tr))

ax1 = Subplot(fig, 1, 1, 1, grid_helper=grid_helper)

fig.add_subplot(ax1)

You may use matplotlib’s Transform instance instead (but a inverse transformation must be defined). Often, coordinate range in a curved coordinate system may have a limited range, or may have cycles. In those cases, a more customized version of grid helper is required.

```
import mpl_toolkits.axisartist.angle_helper as angle_helper

# PolarAxes.PolarTransform takes radian. However, we want our coordinate
```
# system in degree
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

# extreme finder: find a range of coordinate.
# 20, 20: number of sampling points along x, y direction
# The first coordinate (longitude, but theta in polar)
# has a cycle of 360 degree.
# The second coordinate (latitude, but radius in polar) has a minimum of 0
extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
    lon_cycle = 360,
    lat_cycle = None,
    lon_minmax = None,
    lat_minmax = (0, np.inf),
)

# Find a grid values appropriate for the coordinate (degree, 
# minute, second). The argument is a approximate number of grids.
grid_locator1 = angle_helper.LocatorDMS(12)

# And also uses an appropriate formatter. Note that, the
# acceptable Locator and Formatter class is a bit different than
# that of mpl's, and you cannot directly use mpl's Locator and
# Formatter here (but may be possible in the future).
tick_formatter1 = angle_helper.FormatterDMS()

grid_helper = GridHelperCurveLinear(tr,
    extreme_finder=extreme_finder,
    grid_locator1=grid_locator1,
    tick_formatter1=tick_formatter1
)

Again, the transData of the axes is still a rectilinear coordinate (image coordinate). You may manually do conversion between two coordinates, or you may use Parasite Axes for convenience.:

ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

# A parasite axes with given transform
ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anthing you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)
27.2.5 FloatingAxis

A floating axis is an axis one of whose data coordinate is fixed, i.e, its location is not fixed in Axes coordinate but changes as axes data limits changes. A floating axis can be created using `new_floating_axis` method. However, it is your responsibility that the resulting AxisArtist is properly added to the axes. A recommended way is to add it as an item of Axes's axis attribute::

```python
# floating axis whose first (index starts from 0) coordinate (theta) is fixed at 60
ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^\circ$")
axis.label.set_visible(True)
```

See the first example of this page.

27.2.6 Current Limitations and TODO's

The code need more refinement. Here is a incomplete list of issues and TODO’s

- No easy way to support a user customized tick location (for curvilinear grid). A new Locator class needs to be created.
- FloatingAxis may have coordinate limits, e.g., a floating axis of x = 0, but y only spans from 0 to 1.
- The location of axislabel of FloatingAxis needs to be optionally given as a coordinate value. ex, a floating axis of x=0 with label at y=1
28.1 `mpl_toolkits.axes_grid.axes_size`

```python
class mpl_toolkits.axes_grid.axes_size.Fixed(fixed_size)
    Simple fixed size with absolute part = fixed_size and relative part = 0

class mpl_toolkits.axes_grid.axes_size.Scaled(scalable_size)
    Simple scaled(?) size with absolute part = 0 and relative part = scalable_size

class mpl_toolkits.axes_grid.axes_size.AxesX(axes, aspect=1.0, ref_ax=None)
    Scaled size whose relative part corresponds to the data width of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.AxesY(axes, aspect=1.0, ref_ax=None)
    Scaled size whose relative part corresponds to the data height of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.MaxWidth(artist_list)
    Size whose absolute part is the largest width of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.MaxHeight(artist_list)
    Size whose absolute part is the largest height of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.Fraction(fraction, ref_size)
    An instance whose size is a fraction of the ref_size.
    >>> s = Fraction(0.3, AxesX(ax))

class mpl_toolkits.axes_grid.axes_size.Padded(size, pad)
    Return a instance where the absolute part of size is increase by the amount of pad.

mpl_toolkits.axes_grid.axes_size.from_any(size, fraction_ref=None)
    Creates Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %. The second argument is only meaningful when Fraction unit is created.
    >>> a = Size.from_any(1.2) # => Size.Fixed(1.2)
    >>> Size.from_any("50\%", a) # => Size.Fraction(0.5, a)
```
28.2 mpl_toolkits.axes_grid.axes_divider

class mpl_toolkits.axes_grid.axes_divider.Divider(fig, pos, horizontal, vertical, aspect=None, anchor=u'C')

This is the class that is used calculates the axes position. It divides the given rectangular area into several sub-rectangles. You initialize the divider by setting the horizontal and vertical lists of sizes (mpl_toolkits.axes_grid.axes_size) that the division will be based on. You then use the new_locator method to create a callable object that can be used as the axes_locator of the axes.

Parameters

- **fig** – matplotlib figure
- **pos** – position (tuple of 4 floats) of the rectangle that will be divided.
- **horizontal** – list of sizes (axes_size) for horizontal division
- **vertical** – list of sizes (axes_size) for vertical division
- **aspect** – if True, the overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.
- **anchor** – Determine how the reduced rectangle is placed when aspect is True.

add_auto_adjustable_area(use_axes, pad=0.1, adjust_dirs=None)

append_size(position, size)

get_anchor()
    return the anchor

get_aspect()
    return aspect

get_horizontal()
    return horizontal sizes

get_horizontal_sizes(renderer)

get_locator()

get_position()
    return the position of the rectangle.

get_position_runtime(ax, renderer)

get_vertical()
    return vertical sizes

get_vertical_sizes(renderer)
get_vsize_hsize()

locate(nx, ny, nx1=None, ny1=None, axes=None, renderer=None)

Parameters

- **nx1** *(nx,)* – Integers specifying the column-position of the cell. When nx1 is None, a single nx-th column is specified. Otherwise location of columns spanning between nx to nx1 (but excluding nx1-th column) is specified.

- **ny1** *(ny,)* – same as nx and nx1, but for row positions.

new_locator(nx, ny, nx1=None, ny1=None)

returns a new locator *(mpl_toolkits.axes_grid.axes_divider.AxesLocator)* for specified cell.

Parameters

- **nx1** *(nx,)* – Integers specifying the column-position of the cell. When nx1 is None, a single nx-th column is specified. Otherwise location of columns spanning between nx to nx1 (but excluding nx1-th column) is specified.

- **ny1** *(ny,)* – same as nx and nx1, but for row positions.

set_anchor(anchor)

Parameters

- **anchor** – anchor position

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘C’</td>
<td>Center</td>
</tr>
<tr>
<td>‘SW’</td>
<td>bottom left</td>
</tr>
<tr>
<td>‘S’</td>
<td>bottom</td>
</tr>
<tr>
<td>‘SE’</td>
<td>bottom right</td>
</tr>
<tr>
<td>‘E’</td>
<td>right</td>
</tr>
<tr>
<td>‘NE’</td>
<td>top right</td>
</tr>
<tr>
<td>‘N’</td>
<td>top</td>
</tr>
<tr>
<td>‘NW’</td>
<td>top left</td>
</tr>
<tr>
<td>‘W’</td>
<td>left</td>
</tr>
</tbody>
</table>

set_aspect(aspect=False)

Parameters

- **anchor** – True or False

set_horizontal(h)

Parameters

- **horizontal** – list of sizes *(axes_size)* for horizontal division

set_locator(_locator)


set_position(pos)
set the position of the rectangle.

Parameters pos – position (tuple of 4 floats) of the rectangle that will be divided.

set_vertical(v)

Parameters horizontal – list of sizes (axes_size) for horizontal division.

class mpl_toolkits.axes_grid.axes_divider.AxesLocator(axes_divider, nx, ny, nx1=None, ny1=None)
A simple callable object, initialized with AxesDivider class, returns the position and size of the given cell.

Parameters
• axes_divider – An instance of AxesDivider class.
• nx1 (nx) – Integers specifying the column-position of the cell. When nx1 is None, a single nx-th column is specified. Otherwise location of columns spanning between nx to nx1 (but excluding nx1-th column) is is specified.
• ny1 (ny) – same as nx and nx1, but for row positions.

get_subplotspec()

class mpl_toolkits.axes_grid.axes_divider.SubplotDivider(fig, *args, **kwargs)
The Divider class whose rectangle area is specified as a subplot geometry.

fig is a matplotlib.figure.Figure instance.

args is the tuple (numRows, numCols, plotNum), where the array of subplots in the figure has dimensions numRows, numCols, and where plotNum is the number of the subplot being created. plotNum starts at 1 in the upper left corner and increases to the right.

If numRows <= numCols <= plotNum < 10, args can be the decimal integer numRows * 100 + numCols * 10 + plotNum.

change_geometry(numrows, numcols, num)
change subplot geometry, e.g., from 1,1,1 to 2,2,3

get_geometry()
get the subplot geometry, e.g., 2,2,3

get_position()
return the bounds of the subplot box

get_subplotspec()
get the SubplotSpec instance

set_subplotspec(subplotspec)
set the SubplotSpec instance
update_params()
    update the subplot position from fig.subplotpars

class mpl_toolkits.axes_grid.axes_divider.AxesDivider(axes, xref=None, yref=None)
    Divider based on the pre-existing axes.

    Parameters
    axes -- axes

    append_axes(position, size, pad=None, add_to_figure=True, **kwargs)
    create an axes at the given position with the same height (or width) of the main axes.
    position ['left'|'right'|'bottom'|'top']
    size and pad should be axes_grid.axes_size compatible.

    new_horizontal(size, pad=None, pack_start=False, **kwargs)
    Add a new axes on the right (or left) side of the main axes.

    Parameters
    • size -- A width of the axes. A axes_size instance or if float or string is given,
            from_any function is used to create one, with ref_size set to AxesX instance of
            the current axes.
    • pad -- pad between the axes. It takes same argument as size.
    • pack_start -- If False, the new axes is appended at the end of the list, i.e., it
                    became the right-most axes. If True, it is inserted at the start of the list, and
                    becomes the left-most axes.

    All extra keywords arguments are passed to the created axes. If axes_class is given, the new
    axes will be created as an instance of the given class. Otherwise, the same class of the main axes
    will be used.

    new_vertical(size, pad=None, pack_start=False, **kwargs)
    Add a new axes on the top (or bottom) side of the main axes.

    Parameters
    • size -- A height of the axes. A axes_size instance or if float or string is given,
            from_any function is used to create one, with ref_size set to AxesX instance of
            the current axes.
    • pad -- pad between the axes. It takes same argument as size.
    • pack_start -- If False, the new axes is appended at the end of the list, i.e., it
                    became the top-most axes. If True, it is inserted at the start of the list, and
                    becomes the bottom-most axes.

    All extra keywords arguments are passed to the created axes. If axes_class is given, the new
    axes will be created as an instance of the given class. Otherwise, the same class of the main axes
    will be used.
28.3 mpl_toolkits.axes_grid.axes_grid

class mpl_toolkits.axes_grid.axes_grid.Grid(fig, rect, nrows_ncols, ngrids=None, direction=u’row’, axes_pad=0.02, add_all=True, share_all=False, share_x=True, share_y=True, label_mode=u’L’, axes_class=None)

Build an Grid instance with a grid nrows*ncols Axes in Figure fig with rect=[left, bottom, width, height] (in Figure coordinates) or the subplot position code (e.g., “121”).

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>“row”</td>
<td>[“row”</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>share_x</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_y</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[ “L”</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td>a type object which must be a subclass of Axes</td>
</tr>
</tbody>
</table>

class mpl_toolkits.axes_grid.axes_grid.ImageGrid(fig, rect, nrows_ncols, ngrids=None, direction=u’row’, axes_pad=0.02, add_all=True, share_all=False, aspect=True, label_mode=u’L’, cbar_mode=None, cbar_location=u’right’, cbar_pad=None, cbar_size=u’5%’, cbar_set_cax=True, axes_class=None)

Build an ImageGrid instance with a grid nrows*ncols Axes in Figure fig with rect=[left, bottom, width, height] (in Figure coordinates) or the subplot position code (e.g., “121”).

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>“row”</td>
<td>[“row”</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float pad between axes given in inches or tuple-like of floats, (horizontal padding, vertical padding)</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>[ “L”</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[ “each”</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>[ “left”</td>
</tr>
<tr>
<td>cbar_loc</td>
<td>right</td>
<td>[ “left”</td>
</tr>
<tr>
<td>cbar_pad</td>
<td>None</td>
<td>[ True</td>
</tr>
<tr>
<td>cbar_size</td>
<td>“5%”</td>
<td>a type object which must be a subclass of axes_grid’s subclass of Axes</td>
</tr>
<tr>
<td>cbar_set_cax</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td>a type object which must be a subclass of axes_grid’s subclass of Axes</td>
</tr>
</tbody>
</table>

**cbar_set_cax** [if True, each axes in the grid has a cax] attribute that is bind to associated cbar_axes.

### 28.4 `mpl_toolkits.axes_grid.axis_artist`

class `mpl_toolkits.axes_grid.axis_artist.AxisArtist(axes, helper, offset=None, axis_direction=u'bottom', **kw)`

An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, ticklabels, and axis label.

- `axes`: axes
- `helper`: an AxisArtistHelper instance.

**LABELPAD**

**ZORDER = 2.5**

**draw**(*artist, renderer, *args, **kwargs*)

Draw the axis lines, tick lines and labels

**get_axisline_style()**

return the current axisline style.

**get_helper()**

Return axis artist helper instance.

**get_tightbbox**(renderer)

**get_transform()**
invert_ticklabel_direction()

set_axis_direction(axis_direction)

Adjust the direction, text angle, text alignment of ticklabels, labels following the matplotlib convention for the rectangle axes.

The axis_direction must be one of [left, right, bottom, top].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels location</td>
<td>&quot;-&quot;</td>
<td>&quot;-&quot;</td>
<td>&quot;+&quot;</td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td>axislabel location</td>
<td>&quot;-&quot;</td>
<td>&quot;-&quot;</td>
<td>&quot;+&quot;</td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the direction “+” and “-” are relative to the direction of the increasing coordinate. Also, the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_axislabel_direction(label_direction)

Adjust the direction of the axislabel.

ACCEPTS: [ “+” | “-” ]

Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

set_axisline_style(axisline_style=None, **kw)

Set the axisline style.

axisline_style can be a string with axisline style name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_arrowstyle("->",size=1.5") set_arrowstyle("->", size=1.5)

Old attrs simply are forgotten.

Without argument (or with arrowstyle=None), return available styles as a list of strings.

set_label(s)

set_ticklabel_direction(tick_direction)

Adjust the direction of the ticklabel.

ACCEPTS: [ “+” | “-” ]

Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

toggle(all=None, ticks=None, ticklabels=None, label=None)

Toggle visibility of ticks, ticklabels, and (axis) label. To turn all off,
axis.toggle(all=False)

To turn all off but ticks on
axis.toggle(all=False, ticks=True)

To turn all on but (axis) label off
axis.toggle(all=True, label=False))

class mpl_toolkits.axes_grid.axis_artist.Ticks(ticksize, tick_out=False, **kwargs)

Ticks are derived from Line2D, and note that ticks themselves are markers. Thus, you should use set_mec, set_mew, etc.

To change the tick size (length), you need to use set_ticksize. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use set_tick_out(False).

get_tick_out()
Return True if the tick will be rotated by 180 degree.

get_ticksize()
Return length of the ticks in points.

set_tick_out(b)
set True if tick need to be rotated by 180 degree.

set_ticksize(ticksize)
set length of the ticks in points.

class mpl_toolkits.axes_grid.axis_artist.AxisLabel(*kl, **kwargs)

Axis Label. Derived from Text. The position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text.

To change the pad between ticklabels and axis label, use set_pad.

get_pad()
return pad in points. See set_pad for more details.

set_axis_direction(d)
Adjust the text angle and text alignment of axis label according to the matplotlib convention.

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_pad(pad)
Set the pad in points. Note that the actual pad will be the sum of the internal pad and the external pad (that are set automatically by the AxisArtist), and it only set the internal pad

class mpl_toolkits.axes_grid.axis_artist.TickLabels(**kwargs)

Tick Labels. While derived from Text, this single artist draws all ticklabels. As in AxisLabel, the
position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text. Unlike the ticklabels of the mainline matplotlib, properties of single ticklabel alone cannot modified.

To change the pad between ticks and ticklabels, use set_pad.

```python
get_texts_widths_heights_descents(renderer)
```

return a list of width, height, descent for ticklabels.

```python
set_axis_direction(label_direction)
```

Adjust the text angle and text alignment of ticklabels according to the matplotlib convention.

The `label_direction` must be one of [left, right, bottom, top].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the text angles are actually relative to $(90 + \text{angle of the direction to the ticklabel})$, which gives 0 for bottom axis.
Part VI

mplot3d
The mplot3d toolkit adds simple 3D plotting capabilities to matplotlib by supplying an axes object that can create a 2D projection of a 3D scene. The resulting graph will have the same look and feel as regular 2D plots.

29.1 mplot3d tutorial

The interactive backends also provide the ability to rotate and zoom the 3D scene. One can rotate the 3D scene by simply clicking-and-dragging the scene. Zooming is done by right-clicking the scene and dragging the mouse up and down. Note that one does not use the zoom button like one would use for regular 2D plots.
29.1.1 Getting started

An Axes3D object is created just like any other axes using the projection='3d' keyword. Create a new matplotlib.figure.Figure and add a new axes to it of type Axes3D:

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

New in version 1.0.0: This approach is the preferred method of creating a 3D axes.

**Note:** Prior to version 1.0.0, the method of creating a 3D axes was different. For those using older versions of matplotlib, change `ax = fig.add_subplot(111, projection='3d')` to `ax = Axes3D(fig).

29.1.2 Line plots

Axes3D.plot(xs, ys, *args, **kwargs)

Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs, ys</code></td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td><code>zs</code></td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td><code>zdir</code></td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to plot()
29.1.3 Scatter plots

Axes3D.scatter(xs, ys, zs=0, zdir='z', s=20, c='b', depthshade=True, *args, **kwargs)

Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>Positions of data points.</td>
</tr>
<tr>
<td>zs</td>
<td>Either an array of the same length as xs and ys or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>s</td>
<td>Size in points^2. It is a scalar or an array of the same length as x and y.</td>
</tr>
<tr>
<td>c</td>
<td>A color. c can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). Note that c should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. c can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
<tr>
<td>depthshade</td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is True.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to scatter().

Returns a Patch3DCollection
29.1.4 Wireframe plots

Axes3D.plot_wireframe(X, Y, Z, *args, **kwargs)

Plot a 3D wireframe.

The rstride and cstride kwargs set the stride used to sample the input data to generate the graph.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size), defaults to 1</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size), defaults to 1</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to LineCollection.

Returns a Line3DCollection
29.1.5 Surface plots

Axes3D.plot_surface(X, Y, Z, *args, **kwargs)

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the cmap argument.

The rstride and cstride kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size), defaults to 10</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size), defaults to 10</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>facecolors</td>
<td>Face colors for the individual patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>
Other arguments are passed on to `Poly3DCollection`
29.1.6 Tri-Surface plots

Axes3D.plot_trisurf(*args, **kwargs)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a Triangulation object, or:

```
plot_trisurf(X, Y, ...)  
plot_trisurf(X, Y, triangles, ...) 
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See Triangulation for a explanation of these possibilities.
The remaining arguments are:

```
plot_trisurf(..., Z)
```

where \( Z \) is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to `Poly3DCollection`

**Examples:**

```
1.0 0.5 0.0
0.5 1.0 1.0
0.0 0.5 0.0
0.5 1.0 0.6
0.4 0.2 0.0
0.2 0.4 0.6
```

![](image.png)
New in version 1.2.0: This plotting function was added for the v1.2.0 release.
29.1.7 Contour plots

Axes3D.contour(X, Y, Z, *args, **kwargs)

Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to `contour()`

Returns a contour
Chapter 29. Matplotlib mplot3d toolkit
29.1.8 Filled contour plots

Axes3D.contourf(X, Y, Z, *args, **kwargs)
Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to contourf()

Returns a contourf

Changed in version 1.1.0: The zdir and offset kwargs were added.
New in version 1.1.0: The feature demoed in the second contourf3d example was enabled as a result of a bugfix for version 1.1.0.

29.1.9 Polygon plots

Axes3D.add_collection3d(col, zs=0, zdir='z')

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:

- PolyCollection
- LineCollection
- PatchCollection
29.1.10 Bar plots

```python
Axes3D.bar(left, height, zs=0, zdir='z', *args, **kwargs)
```

Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td>height</td>
<td>The height of the bars.</td>
</tr>
<tr>
<td>zs</td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (’x’, ’y’ or ’z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`
29.1.11 Quiver

Axes3D.quiver(*args, **kwargs)

Plot a 3D field of arrows.

call signatures:
quiver(X, Y, Z, U, V, W, **kwargs)

Arguments:

X, Y, Z: The x, y and z coordinates of the arrow locations

U, V, W: The direction vector that the arrow is pointing

The arguments could be array-like or scalars, so long as they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

length: [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes

arrow_length_ratio: [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3
Any additional keyword arguments are delegated to LineCollection
29.1.12 2D plots in 3D

29.1.13 Text

Axes3D.text(x, y, z, zs=0, zdir=z, **kwargs)

Add text to the plot. kwargs will be passed on to Axes.text, except for the zdir keyword, which sets the direction to be used as the z direction.
29.1.14 Subplotting

Having multiple 3D plots in a single figure is the same as it is for 2D plots. Also, you can have both 2D and 3D plots in the same figure.

New in version 1.0.0: Subplotting 3D plots was added in v1.0.0. Earlier version can not do this.
29.2 mplot3d API
Contents

- mplot3d API
  - axes3d
  - axis3d
  - art3d
  - proj3d

29.2.1 axes3d

Note: Significant effort went into bringing axes3d to feature-parity with regular axes objects for version 1.1.0. However, more work remains. Please report any functions that do not behave as expected as a bug. In addition, help and patches would be greatly appreciated!

Module containing Axes3D, an object which can plot 3D objects on a 2D matplotlib figure.

class mpl_toolkits.mplot3d.axes3d.Axes3D(fig, rect=None, *args, **kwargs)

  Bases: matplotlib.axes._axes.Axes

  3D axes object.

  add_collection3d(col, zs=0, zdir='z')
  
  Add a 3D collection object to the plot.

  2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

  Supported are:

  - PolyCollection
  - LineCollection
  - PatchCollection

  add_contour_set(cset, extend3d=False, stride=5, zdir='z', offset=None)

  add_contourf_set(cset, zdir='z', offset=None)

  auto_scale_xyz(X, Y, Z=None, had_data=None)

  autoscale(enable=True, axis='both', tight=None)

  Convenience method for simple axis view autoscaling. See matplotlib.axes.Axes.autoscale() for full explanation. Note that this function behaves the same, but for all three axes. Therefore, ‘z’ can be passed for axis, and ‘both’ applies to all three axes.

  New in version 1.1.0: This function was added, but not tested. Please report any bugs.
autoscale_view(tight=None, scalex=True, scaley=True, scalez=True)
Autoscale the view limits using the data limits. See
matplotlib.axes.Axes.autoscale_view() for documentation. Note that this func-
tion applies to the 3D axes, and as such adds the scalez to the function arguments.

Changed in version 1.1.0: Function signature was changed to better match the 2D version. tight
is now explicitly a kwarg and placed first.

Changed in version 1.2.1: This is now fully functional.

bar(left, height, zs=0, zdir='z', *args, **kwargs)
Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td>height</td>
<td>The height of the bars.</td>
</tr>
<tr>
<td>zs</td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto bar().

Returns a Patch3DCollection

bar3d(x, y, z, dx, dy, dz, color='b', zsort='average', *args, **kwargs)
Generate a 3D bar, or multiple bars.

When generating multiple bars, x, y, z have to be arrays. dx, dy, dz can be arrays or scalars.

color can be:

- A single color value, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

Keyword arguments are passed onto Poly3DCollection()

can_pan()
Return True if this axes supports the pan/zoom button functionality.

3D axes objects do not use the pan/zoom button.
can_zoom()
Return True if this axes supports the zoom box button functionality.

3D axes objects do not use the zoom box button.

cla()
Clear axes

clabel(*args, **kwargs)
This function is currently not implemented for 3D axes. Returns None.

contour(X, Y, Z, *args, **kwargs)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to contour()

Returns a contour

contour3D(X, Y, Z, *args, **kwargs)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to contour()

Returns a contour

contourf(X, Y, Z, *args, **kwargs)
Create a 3D contourf plot.
Matplotlib, Release 1.4.3

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to `contourf()`

Returns a contourf

Changed in version 1.1.0: The zdir and offset kwargs were added.

**contourf3D(X, Y, Z, *args, **kwargs)**

Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to `contourf()`

Returns a contourf

Changed in version 1.1.0: The zdir and offset kwargs were added.

**convert_zunits(z)**

For artists in an axes, if the zaxis has units support, convert z using zaxis unit type

New in version 1.2.1.

**disable_mouse_rotation()**

Disable mouse button callbacks.

**draw(renderer)**

**format_coord(xd, yd)**

Given the 2D view coordinates attempt to guess a 3D coordinate. Looks for the nearest edge to the point and then assumes that the point is at the same z location as the nearest point on the edge.

**format_zdata(z)**

Return z string formatted. This function will use the fmt_zdata attribute if it is callable, else will fall back on the zaxis major formatter

**get_autoscale_on()**

Get whether autoscaling is applied for all axes on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.
get_autoscalez_on()
Get whether autoscaling for the z-axis is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_axis_position()

get_axisbelow()
Get whether axis below is true or not.

For axes3d objects, this will always be True

New in version 1.1.0: This function was added for completeness.

get_children()

get_frame_on()
Get whether the 3D axes panels are drawn

New in version 1.1.0.

get_proj()
Create the projection matrix from the current viewing position.

elev stores the elevation angle in the z plane
azim stores the azimuth angle in the x,y plane
dist is the distance of the eye viewing point from the object point.

get_w_lims()
Get 3D world limits.

get_xlim()
Get the x-axis range [left, right]

Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_xlim3d()
Get the x-axis range [left, right]

Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_ylim()
Get the y-axis range [bottom, top]

Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_ylim3d()
Get the y-axis range [bottom, top]

Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_zbound()
Returns the z-axis numerical bounds where:

lowerBound < upperBound

New in version 1.1.0: This function was added, but not tested. Please report any bugs.
get_zlabel()
Get the z-label text string.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlim()
Get 3D z limits.

get_zlim3d()
Get 3D z limits.

get_zmajorticklabels()
Get the ztick labels as a list of Text instances

New in version 1.1.0.

get_zminorticklabels()
Get the ztick labels as a list of Text instances

Note: Minor ticks are not supported. This function was added only for completeness.

New in version 1.1.0.

get_zscale()

get_zticklabels(minor=False)
Get ztick labels as a list of Text instances. See matplotlib.axes.Axes.get_yticklabels() for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

get_zticklines()
Get ztick lines as a list of Line2D instances. Note that this function is provided merely for completeness. These lines are re-calculated as the display changes.

New in version 1.1.0.

get_zticks(minor=False)
Return the z ticks as a list of locations See matplotlib.axes.Axes.get_yticks() for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

grid(b=True, **kwargs)
Set / unset 3D grid.

Note: Currently, this function does not behave the same as matplotlib.axes.Axes.grid(), but it is intended to eventually support that behavior.
have_units()
Return True if units are set on the x, y, or z axes

invert_zaxis()
Invert the z-axis.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

locator_params(axis='both', tight=None, **kwargs)
Convenience method for controlling tick locators.
See matplotlib.axes.Axes.locator_params() for full documentation Note that this is for Axes3D objects, therefore, setting axis to ‘both’ will result in the parameters being set for all three axes. Also, axis can also take a value of ‘z’ to apply parameters to the z axis.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

margins(*args, **kw)
Convenience method to set or retrieve autoscaling margins.
signatures:: margins()
returns xmin, ymin, zmin
margins(margin)
margins(xmin, ymin, zmin)
margins(x=xmin, y=ymin, z=zmin)
margins(..., tight=False)

All forms above set the xmin, ymin and zmin parameters. All keyword parameters are optional. A single argument specifies xmin, ymin and zmin. The tight parameter is passed to autoscale_view(), which is executed after a margin is changed; the default here is True, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting tight to None will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if xmin is not None, then xmin times the X data interval will be added to each end of that interval before it is used in autoscaling.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

mouse_init(rotate_btn=1, zoom_btn=3)
Initializes mouse button callbacks to enable 3D rotation of the axes. Also optionally sets the mouse buttons for 3D rotation and zooming.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate_btn</td>
<td>The integer or list of integers specifying which mouse button or buttons to use for 3D rotation of the axes. Default = 1.</td>
</tr>
<tr>
<td>zoom_btn</td>
<td>The integer or list of integers specifying which mouse button or buttons to use to zoom the 3D axes. Default = 3.</td>
</tr>
</tbody>
</table>

name = u’3d’

```python
plot(xs, ys, *args, **kwargs)
```
Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to `plot()`

```python
plot3D(xs, ys, *args, **kwargs)
```
Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to `plot()`

```python
plot_surface(X, Y, Z, *args, **kwargs)
```
Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the `cmap` argument.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size), defaults to 10</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size), defaults to 10</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>facecolors</td>
<td>Face colors for the individual patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

Other arguments are passed on to `Poly3DCollection`
plot_trisurf(*args, **kwargs)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

The (optional) triangulation can be specified in one of two ways; either:

plot_trisurf(triangulation, ...)

where triangulation is a Triangulation object, or:

plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)

in which case a Triangulation object will be created. See Triangulation for a explanation of these possibilities.

The remaining arguments are:

plot_trisurf(..., Z)

where Z is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to Poly3DCollection

Examples:
29.2. mplot3d API
New in version 1.2.0: This plotting function was added for the v1.2.0 release.

\texttt{plot\_wireframe}(X, \ Y, \ Z, \ *\textit{args}, \ **\textit{kwargs})

Plot a 3D wireframe.

The \texttt{rstride} and \texttt{cstride} \textit{kwargs} set the stride used to sample the input data to generate the graph.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{X, Y, Z}</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>\texttt{rstride}</td>
<td>Array row stride (step size), defaults to 1</td>
</tr>
<tr>
<td>\texttt{cstride}</td>
<td>Array column stride (step size), defaults to 1</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to \texttt{LineCollection}.

Returns a \texttt{Line3DCollection}

\texttt{quiver}(\*\textit{args}, \ **\textit{kwargs})

Plot a 3D field of arrows.

call signatures:

\texttt{quiver}(X, \ Y, \ Z, \ U, \ V, \ W, \ **\textit{kwargs})

Arguments:

\texttt{X, Y, Z}: The x, y and z coordinates of the arrow locations
\( U, V, W: \) The direction vector that the arrow is pointing

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

- **length:** [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes
- **arrow_length_ratio:** [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3

Any additional keyword arguments are delegated to LineCollection

```python
quiver3D(**kwargs)
```

Plot a 3D field of arrows.

call signatures:

```python
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

- **X, Y, Z:** The x, y and z coordinates of the arrow locations
- **U, V, W:** The direction vector that the arrow is pointing

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

- **length:** [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes
- **arrow_length_ratio:** [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3

Any additional keyword arguments are delegated to LineCollection

```python
scatter(xs, ys, zs=0, zdir='z', s=20, c=u'b', depthshade=True, **kwargs)
```

Create a scatter plot.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>Positions of data points. Either an array of the same length as xs and ys or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>zs</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>zdir</td>
<td>size in points^2. It is a scalar or an array of the same length as x and y.</td>
</tr>
<tr>
<td>s</td>
<td>a color. c can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). Note that c should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. c can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
<tr>
<td>c</td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is True.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to scatter().

Returns a Patch3DCollection

```
scatter3D(xs, ys, zs=0, zdir='z', s=20, c='b', depthshade=True, *args, **kwargs)
```

Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>Positions of data points. Either an array of the same length as xs and ys or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>zs</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>zdir</td>
<td>size in points^2. It is a scalar or an array of the same length as x and y.</td>
</tr>
<tr>
<td>s</td>
<td>a color. c can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). Note that c should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. c can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
<tr>
<td>c</td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is True.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to scatter().

Returns a Patch3DCollection

```
set_autoscale_on(b)
```

Set whether autoscaling is applied on plot commands

accepts: [ True | False ]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```
set_autoscalez_on(b)
```

Set whether autoscaling for the z-axis is applied on plot commands
accepts: [ True | False ]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```python
set_axis_off()
```

```python
set_axis_on()
```

```python
set_axisbelow(b)
```

Set whether the axis ticks and gridlines are above or below most artists

For axes3d objects, this will ignore any settings and just use True

ACCEPTS: [ True | False ]

New in version 1.1.0: This function was added for completeness.

```python
set_frame_on(b)
```

Set whether the 3D axes panels are drawn

ACCEPTS: [ True | False ]

New in version 1.1.0.

```python
set_title(label, fontdict=None, loc='center', **kwargs)
```

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

- **label** : str
  - Text to use for the title

- **fontdict** : dict
  - A dictionary controlling the appearance of the title text, the default fontdict is:

    ```python
    {'fontsize': rcParams['axes.titlesize'],
    'fontweight': rcParams['axes.titleweight'],
    'verticalalignment': 'baseline',
    'horizontalalignment': loc}
    ```

- **loc** : {'center', 'left', 'right'}, str, optional
  - Which title to set, defaults to 'center'

**Returns**

- **text** : Text
  - The matplotlib text instance representing the title

**Other Parameters**

- **kwars** : text properties
  - Other keyword arguments are text properties, see Text for a list of valid text properties.
set_top_view()

set_xlim((left=None, right=None, emit=True, auto=False, **kw))
Set 3D x limits.
See matplotlib.axes.Axes.set_xlim() for full documentation.

set_xlim3d((left=None, right=None, emit=True, auto=False, **kw))
Set 3D x limits.
See matplotlib.axes.Axes.set_xlim() for full documentation.

set_xscale(value, **kwargs)
Call signature:

set_xscale(value)
Set the scaling of the x-axis: u'linear' | u'log' | u'symlog'
ACCEPTS: [u'linear' | u'log' | u'symlog']
Different kwargs are accepted, depending on the scale: 'linear'
'log'
   basex/basey: The base of the logarithm
   nonposx/nonposy: ['mask' | 'clip'] non-positive values in x or y can be masked as in-
                   valid, or clipped to a very small positive number
   subsx/subsy: Where to place the subticks between each major tick. Should be a se-
                quence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]
                will place 8 logarithmically spaced minor ticks between each major tick.
'symlog'
   basex/basey: The base of the logarithm
   linthreshx/linthreshy: The range (-x, x) within which the plot is linear (to avoid having
                          the plot go to infinity around zero).
   subsx/subsy: Where to place the subticks between each major tick. Should be a se-
                quence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]
                will place 8 logarithmically spaced minor ticks between each major tick.
   linscalex/linscaley: This allows the linear range (-linthresh to linthresh) to be stretched
                        relative to the logarithmic range. Its value is the number of decades to use for each
                        half of the linear range. For example, when linscale == 1.0 (the default), the space
                        used for the positive and negative halves of the linear range will be equal to one
                        decade in the logarithmic range.

   New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_ylim((bottom=None, top=None, emit=True, auto=False, **kw))
Set 3D y limits.
See matplotlib.axes.Axes.set_ylim() for full documentation.

set_ylim3d((bottom=None, top=None, emit=True, auto=False, **kw))
Set 3D y limits.
See `matplotlib.axes.Axes.set_yscale()` for full documentation.

**set_yscale(value, **kwargs)**

Call signature:

```python
set_yscale(value)
```

Set the scaling of the y-axis: u'linear' | u'log' | u'symlog'

**ACCEPTS:** [u'linear' | u'log' | u'symlog']

**Different kwargs are accepted, depending on the scale:** ‘linear’

‘log’

- **basex/basey:** The base of the logarithm
- **nonposx/nonposy:** ['mask' | 'clip'] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
- **subsx/subsy:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

‘symlog’

- **basex/basey:** The base of the logarithm
- **linthreshx/linthreshy:** The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
- **subsx/subsy:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

- **linscalex/linscaley:** This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `linscale` == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zbound(lower=None, upper=None)**

Set the lower and upper numerical bounds of the z-axis. This method will honor axes inversion regardless of parameter order. It will not change the `_autoscaleZon` attribute.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zlabel(zlabel, fontdict=None, labelpad=None, **kwargs)**

Set zlabel. See doc for `set_ylabel()` for description.

**Note:** Currently, `labelpad` does not have an effect on the labels.

**set_zlim(bottom=None, top=None, emit=True, auto=False, **kw)**

Set 3D z limits.

See `matplotlib.axes.Axes.set_zlim()` for full documentation.
set_zlim3d(\(bottom=None, top=None, emit=True, auto=False, **kw)\)
Set 3D z limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation

set_zmargin(\(m)\)
Set padding of Z data limits prior to autoscaling.

\(m\) times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zscale(value, **kwargs)
call signature:

set_zscale(value)

Set the scaling of the z-axis: u'linear' | u'log' | u'symlog'

ACCEPTS: [u'linear' | u'log' | u'symlog']

Different kwargs are accepted, depending on the scale: ‘linear’

‘log’

\(basex/basey\): The base of the logarithm

\(nonposx/nonposy\): [‘mask’ | ‘clip’] non-positive values in \(x\) or \(y\) can be masked as invalid, or clipped to a very small positive number

\(subsx/subsy\): Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘symlog’

\(basex/basey\): The base of the logarithm

\(linthreshx/linthreshy\): The range (\(-x, x\)) within which the plot is linear (to avoid having the plot go to infinity around zero).

\(subsx/subsy\): Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

\(linscalex/linscaley\): This allows the linear range (\(-linthresh to linthresh\)) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when linscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

**Note:** Currently, Axes3D objects only supports linear scales. Other scales may or may not work, and support for these is improving with each release.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zticklabels(*args, **kwargs)
Set z-axis tick labels. See `matplotlib.axes.Axes.set_yticklabels()` for more details.
**Note:** Minor ticks are not supported by Axes3D objects.

New in version 1.1.0.

```
set_zticks(*args, **kwargs)
```

Set z-axis tick locations. See `matplotlib.axes.Axes.set_yticks()` for more details.

**Note:** Minor ticks are not supported.

New in version 1.1.0.

```
text(x, y, z, s, zdir=None, **kwargs)
```

Add text to the plot. kwargs will be passed on to Axes.text, except for the `zdir` keyword, which sets the direction to be used as the z direction.

```
text2D(x, y, s, fontdict=None, withdash=False, **kwargs)
```

Add text to the axes.

Add text in string `s` to axis at location `x`, `y`, data coordinates.

**Parameters**
- `x`, `y` : scalars
  data coordinates
- `s` : string
  text
- `fontdict` : dictionary, optional, default: None
  A dictionary to override the default text properties. If `fontdict` is None, the defaults are determined by your rc parameters.
- `withdash` : boolean, optional, default: False
  Creates a `TextWithDash` instance instead of a `Text` instance.

**Other Parameters**
- `kwargs` : `Text` properties.
  Other miscellaneous text parameters.

**Examples**

Individual keyword arguments can be used to override any given parameter:

```python
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```python
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...       verticalalignment='center',
...       transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `bbox`. `bbox` is a dictionary of `Rectangle` properties. For example:
text3D(x, y, z, s, zdir=None, **kwargs)

Add text to the plot. kwargs will be passed on to Axes.text, except for the zdir keyword, which sets the direction to be used as the z direction.

tick_params(axis='both', **kwargs)

Convenience method for changing the appearance of ticks and tick labels.

See matplotlib.axes.Axes.tick_params() for more complete documentation.

The only difference is that setting axis to ‘both’ will mean that the settings are applied to all three axes. Also, the axis parameter also accepts a value of ‘z’, which would mean to apply to only the z-axis.

Also, because of how Axes3D objects are drawn very differently from regular 2D axes, some of these settings may have ambiguous meaning. For simplicity, the ‘z’ axis will accept settings as if it was like the ‘y’ axis.

Note: While this function is currently implemented, the core part of the Axes3D object may ignore some of these settings. Future releases will fix this. Priority will be given to those who file bugs.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

ticklabel_format(**kwargs)

Convenience method for manipulating the ScalarFormatter used by default for linear axes in Axes3D objects.

See matplotlib.axes.Axes.ticklabel_format() for full documentation. Note that this version applies to all three axes of the Axes3D object. Therefore, the axis argument will also accept a value of ‘z’ and the value of ‘both’ will apply to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

tricontour(*args, **kwargs)

Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to tricontour()

Returns a contour

Changed in version 1.3.0: Added support for custom triangulations
EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

**tricontourf**(*args, **kwargs*)
Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir, offset</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td></td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to **tricontour()**

Returns a **contour**

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

**tunit_cube**(vals=None, M=None)**

**tunit_edges**(vals=None, M=None)**

**unit_cube**(vals=None)**

**update_datalim**(xys, **kwargs)**

**view_init**(elev=None, azim=None)**
Set the elevation and azimuth of the axes.

This can be used to rotate the axes programatically.

‘elev’ stores the elevation angle in the z plane. ‘azim’ stores the azimuth angle in the x,y plane.

if elev or azim are None (default), then the initial value is used which was specified in the Axes3D constructor.

**zaxis_date**(tz=None)**
Sets up z-axis ticks and labels that treat the z data as dates.

tz is a timezone string or tzinfo instance. Defaults to rc value.

**Note:** This function is merely provided for completeness. Axes3D objects do not officially support dates for ticks, and so this may or may not work as expected.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.
zaxis_inverted()
Returns True if the z-axis is inverted.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

mpl_toolkits.mplot3d.axes3d.get_test_data(delta=0.05)
Return a tuple X, Y, Z with a test data set.

mpl_toolkits.mplot3d.axes3d.unit_bbox()

29.2.2 axis3d

Note: Historically, axis3d has suffered from having hard-coded constants controlling the look and feel of the 3D plot. This precluded user level adjustments such as label spacing, font colors and panel colors. For version 1.1.0, these constants have been consolidated into a single private member dictionary, self._axinfo, for the axis object. This is intended only as a stop-gap measure to allow user-level customization, but it is not intended to be permanent.

class mpl_toolkits.mplot3d.axes3d.Axis(adir, v_intervalx, d_intervalx, axes, *args,**kwargs)

Bases: matplotlib.axis.XAxis
draw(renderer)
draw_pane(renderer)
get_major_ticks(numticks=None)
get_rotate_label(text)
get_tick_positions()
get_tightbbox(renderer)
get_view_interval()
    return the Interval instance for this 3d axis view limits
init3d()

set_pane_color(color)
    Set pane color to a RGBA tuple
set_pane_pos(xys)
**set_rotate_label** *(val)*
Whether to rotate the axis label: True, False or None. If set to None the label will be rotated if longer than 4 chars.

**set_view_interval** *(vmin, vmax, ignore=False)*

class **mpl_toolkits.mplot3d.axis3d.XAxis**(adir, *v_intervalx, d_intervalx, axes, *args, **kwargs)
Bases: **mpl_toolkits.mplot3d.axis3d.Axis**

**get_data_interval**()
return the Interval instance for this axis data limits
class **mpl_toolkits.mplot3d.axis3d.YAxis**(adir, *v_intervalx, d_intervalx, axes, *args, **kwargs)
Bases: **mpl_toolkits.mplot3d.axis3d.Axis**

**get_data_interval**()
return the Interval instance for this axis data limits
class **mpl_toolkits.mplot3d.axis3d.ZAxis**(adir, *v_intervalx, d_intervalx, axes, *args, **kwargs)
Bases: **mpl_toolkits.mplot3d.axis3d.Axis**

**get_data_interval**()
return the Interval instance for this axis data limits

**mpl_toolkits.mplot3d.axis3d.get_flip_min_max**(coord, index, mins, maxs)

**mpl_toolkits.mplot3d.axis3d.move_from_center**(coord, centers, deltas, axmask=(True, True, True))
Return a coordinate that is moved by “deltas” away from the center.

**mpl_toolkits.mplot3d.axis3d.tick_update_position**(tick, tickxs, tickys, labelpos)
Update tick line and label position and style.

### 29.2.3 art3d

Module containing 3D artist code and functions to convert 2D artists into 3D versions which can be added to an Axes3D.

class **mpl_toolkits.mplot3d.art3d.Line3D**(xs, ys, zs, *args, **kwargs)
Bases: **matplotlib.lines.Line2D**

3D line object.
Keyword arguments are passed onto Line2D().

draw(renderer)

**set_3d_properties**(zs=0, zdir=’z’)

---

**Matplotlib, Release 1.4.3**

---
class mpl_toolkits.mplot3d.art3d.Line3DCollection(segments, *args, **kwargs)
    Bases: matplotlib.collections.LineCollection
    A collection of 3D lines.
    Keyword arguments are passed onto LineCollection().

do_3d_projection(renderer)
    Project the points according to renderer matrix.

draw(renderer, project=False)

set_segments(segments)
    Set 3D segments

set_sort_zpos(val)
    Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.Patch3D(*args, **kwargs)
    Bases: matplotlib.patches.Patch
    3D patch object.

do_3d_projection(renderer)

draw(renderer)

get_facecolor()

get_path()

set_3d_properties(verts, zs=0, zdir='z')

class mpl_toolkits.mplot3d.art3d.Patch3DCollection(*args, **kwargs)
    Bases: matplotlib.collections.PatchCollection
    A collection of 3D patches.
    Create a collection of flat 3D patches with its normal vector pointed in zdir direction, and located at zs on the zdir axis. ‘zs’ can be a scalar or an array-like of the same length as the number of patches in the collection.
    Constructor arguments are the same as for PatchCollection. In addition, keywords zs=0 and zdir='z' are available.
    Also, the keyword argument “depthshade” is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is True). This is typically desired in scatter plots.

do_3d_projection(renderer)


```python
set_3d_properties(zs, zdir)
```

```python
set_sort_zpos(val)
   Set the position to use for z-sorting.
```

class mpl_toolkits.mplot3d.art3d.Path3DCollection(*args, **kwargs)
   Bases: matplotlib.collections.PathCollection

A collection of 3D paths.

Create a collection of flat 3D paths with its normal vector pointed in `zdir` direction, and located at `zs` on the `zdir` axis. `zs` can be a scalar or an array-like of the same length as the number of paths in the collection.

Constructor arguments are the same as for `PathCollection`. In addition, keywords `zs=0` and `zdir='z'` are available.

Also, the keyword argument “depthshade” is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is `True`). This is typically desired in scatter plots.

```python
do_3d_projection(renderer)
```

```python
set_3d_properties(zs, zdir)
```

```python
set_sort_zpos(val)
   Set the position to use for z-sorting.
```

class mpl_toolkits.mplot3d.art3d.PathPatch3D(path, **kwargs)
   Bases: mpl_toolkits.mplot3d.art3d.Patch3D

3D PathPatch object.

```python
do_3d_projection(renderer)
```

```python
set_3d_properties(path, zs=0, zdir='z')
```

class mpl_toolkits.mplot3d.art3d.Poly3DCollection(verts, *args, **kwargs)
   Bases: matplotlib.collections.PolyCollection

A collection of 3D polygons.

Create a Poly3DCollection.

`verts` should contain 3D coordinates.

Keyword arguments: zsort, see set_zsort for options.

Note that this class does a bit of magic with the _facecolors and _edgecolors properties.

```python
do_3d_projection(renderer)
   Perform the 3D projection for this object.
```
draw(renderer)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_vector(segments3d)
  Optimize points for projection

set_3d_properties()

set_alpha(alpha)
  Set the alpha transparency of the collection. alpha must be a float or None.
  ACCEPTS: float or None

set_edgecolor(colors)

set_edgecolors(colors)

set_facecolor(colors)

set_facecolors(colors)

set_sort_zpos(val)
  Set the position to use for z-sorting.

set_verts(verts, closed=True)
  Set 3D vertices.

set_zsort(zsort)
  Set z-sorting behaviour: boolean: if True use default ‘average’ string: ‘average’, ‘min’ or ‘max’

class mpl_toolkits.mplot3d.art3d.Text3D(x=0, y=0, z=0, text=u'', zdir=u'z', **kwargs)
  Bases: matplotlib.text.Text

  Text object with 3D position and (in the future) direction.

  x, y, z Position of text text Text string to display zdir Direction of text

  Keyword arguments are passed onto Text().
```python
draw(renderer)

set_3d_properties(z=0, zdir=u'z')
```

```python
cmpl_toolkits.mplot3d.art3d.get_colors(c, num)
```
Stretch the color argument to provide the required number num

```python
cmpl_toolkits.mplot3d.art3d.get_dir_vector(zdir)
```

```python
cmpl_toolkits.mplot3d.art3d.get_patch_verts(patch)
```
Return a list of vertices for the path of a patch.

```python
cmpl_toolkits.mplot3d.art3d.iscolor(c)
```

```python
cmpl_toolkits.mplot3d.art3d.juggle_axes(xs, ys, zs, zdir)
```
Reorder coordinates so that 2D xs, ys can be plotted in the plane orthogonal to zdir. zdir is normally x, y or z. However, if zdir starts with a '-' it is interpreted as a compensation for rotate_axes.

```python
cmpl_toolkits.mplot3d.art3d.line_2d_to_3d(line, zs=0, zdir=u'z')
```
Convert a 2D line to 3D.

```python
cmpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d(col, zs=0, zdir=u'z')
```
Convert a LineCollection to a Line3DCollection object.

```python
cmpl_toolkits.mplot3d.art3d.norm_angle(a)
```
Return angle between -180 and +180

```python
cmpl_toolkits.mplot3d.art3d.norm_text_angle(a)
```
Return angle between -90 and +90

```python
cmpl_toolkits.mplot3d.art3d.patch_2d_to_3d(patch, z=0, zdir=u'z')
```
Convert a Patch to a Patch3D object.

```python
cmpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d(col, zs=0, zdir=u'z', depthshade=True)
```
Convert a PatchCollection into a Patch3DCollection object (or a PathCollection into a Path3DCollection object).

Keywords:

- **za** The location or locations to place the patches in the collection along the zdir axis. Defaults to 0.
- **zdir** The axis in which to place the patches. Default is “z”.
- **depthshade** Whether to shade the patches to give a sense of depth. Defaults to True.

```python
cmpl_toolkits.mplot3d.art3d.path_to_3d_segment(path, zs=0, zdir=u'z')
```
Convert a path to a 3D segment.

```python
cmpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d(pathpatch, z=0, zdir=u'z')
```
Convert a PathPatch to a PathPatch3D object.
mpl_toolkits.mplot3d.art3d.paths_to_3d_segments(paths, zs=0, zdir='z')
Convert paths from a collection object to 3D segments.

mpl_toolkits.mplot3d.art3d.poly_collection_2d_to_3d(col, zs=0, zdir='z')
Convert a PolyCollection to a Poly3DCollection object.

mpl_toolkits.mplot3d.art3d.rotate_axes(xs, ys, zs, zdir)
Reorder coordinates so that the axes are rotated with zdir along the original z axis. Prepending the axis with a '-' does the inverse transform, so zdir can be x, -x, y, -y, z or -z.

mpl_toolkits.mplot3d.art3d.text_2d_to_3d(obj, z=0, zdir='z')
Convert a Text to a Text3D object.

mpl_toolkits.mplot3d.art3d.zalpha(colors, zs)
Modify the alphas of the color list according to depth.

29.2.4 proj3d
Various transforms used for by the 3D code

mpl_toolkits.mplot3d.proj3d.inv_transform(xs, ys, zs, M)

mpl_toolkits.mplot3d.proj3d.line2d(p0, p1)
Return 2D equation of line in the form ax+by+c = 0

mpl_toolkits.mplot3d.proj3d.line2d_dist(l, p)
Distance from line to point line is a tuple of coefficients a,b,c.

mpl_toolkits.mplot3d.proj3d.line2d_seg_dist(p1, p2, p0)
distance(s) from line defined by p1 - p2 to point(s) p0
p0[0] = x(s) p0[1] = y(s)
intersection point p = p1 + u*(p2-p1) and intersection point lies within segment if u is between 0 and 1.

mpl_toolkits.mplot3d.proj3d.mod(v)
3d vector length

mpl_toolkits.mplot3d.proj3d.persp_transformation(zfront, zback)

mpl_toolkits.mplot3d.proj3d.proj_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_trans_clip_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_trans_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_transform(xs, ys, zs, M)
Transform the points by the projection matrix.
mpl_toolkits.mplot3d.proj3d.proj_transform_clip(xs, ys, zs, M)
    Transform the points by the projection matrix and return the clipping result returns txs, tys, tzs, tis
mpl_toolkits.mplot3d.proj3d.proj_transform_vec(vec, M)

mpl_toolkits.mplot3d.proj3d.proj_transform_vec_clip(vec, M)

mpl_toolkits.mplot3d.proj3d.rot_x(V, alpha)

mpl_toolkits.mplot3d.proj3d.test_lines_dists()

mpl_toolkits.mplot3d.proj3d.test_proj()

mpl_toolkits.mplot3d.proj3d.test_proj_draw_axes(M, s=1)

mpl_toolkits.mplot3d.proj3d.test_proj_make_M(E=None)

mpl_toolkits.mplot3d.proj3d.test_rot()

mpl_toolkits.mplot3d.proj3d.test_world()

mpl_toolkits.mplot3d.proj3d.transform(xs, ys, zs, M)
    Transform the points by the projection matrix
mpl_toolkits.mplot3d.proj3d.vec_pad_ones(xs, ys, zs)

mpl_toolkits.mplot3d.proj3d.view_transformation(E, R, V)

mpl_toolkits.mplot3d.proj3d.world_transformation(xmin, xmax, ymin, ymax, zmin, zmax)

### 29.3 mplot3d FAQ

#### 29.3.1 How is mplot3d different from MayaVi?

**MayaVi2** is a very powerful and featureful 3D graphing library. For advanced 3D scenes and excellent rendering capabilities, it is highly recommended to use MayaVi2.

mplot3d was intended to allow users to create simple 3D graphs with the same “look-and-feel” as matplotlib’s 2D plots. Furthermore, users can use the same toolkit that they are already familiar with to generate both their 2D and 3D plots.
29.3.2 My 3D plot doesn’t look right at certain viewing angles

This is probably the most commonly reported issue with mplot3d. The problem is that – from some viewing angles – a 3D object would appear in front of another object, even though it is physically behind it. This can result in plots that do not look “physically correct.”

Unfortunately, while some work is being done to reduce the occurrence of this artifact, it is currently an intractable problem, and can not be fully solved until matplotlib supports 3D graphics rendering at its core.

The problem occurs due to the reduction of 3D data down to 2D + z-order scalar. A single value represents the 3rd dimension for all parts of 3D objects in a collection. Therefore, when the bounding boxes of two collections intersect, it becomes possible for this artifact to occur. Furthermore, the intersection of two 3D objects (such as polygons or patches) can not be rendered properly in matplotlib’s 2D rendering engine.

This problem will likely not be solved until OpenGL support is added to all of the backends (patches are greatly welcomed). Until then, if you need complex 3D scenes, we recommend using MayaVi.

29.3.3 I don’t like how the 3D plot is laid out, how do I change that?

Historically, mplot3d has suffered from a hard-coding of parameters used to control visuals such as label spacing, tick length, and grid line width. Work is being done to eliminate this issue. For matplotlib v1.1.0, there is a semi-official manner to modify these parameters. See the note in the axis3d section of the mplot3d API documentation for more information.
Part VII

Toolkits
Toolkits are collections of application-specific functions that extend matplotlib.
30.1 Basemap

(Not distributed with matplotlib)

Plots data on map projections, with continental and political boundaries, see basemap docs.

contour lines over filled continent background
30.2 Cartopy

*(Not distributed with matplotlib)*

An alternative mapping library written for matplotlib v1.2 and beyond. Cartopy builds on top of matplotlib to provide object oriented map projection definitions and close integration with Shapely for powerful yet easy-to-use vector data processing tools. An example plot from the Cartopy gallery:

US States which intersect the track of Hurricane Katrina (2005)
31.1 mplot3d

31.1.1 mplot3d

Matplotlib mplot3d toolkit

The mplot3d toolkit adds simple 3D plotting capabilities to matplotlib by supplying an axes object that can create a 2D projection of a 3D scene. The resulting graph will have the same look and feel as regular 2D plots.
The interactive backends also provide the ability to rotate and zoom the 3D scene. One can rotate the 3D scene by simply clicking-and-dragging the scene. Zooming is done by right-clicking the scene and dragging the mouse up and down. Note that one does not use the zoom button like one would use for regular 2D plots.

mplot3d tutorial

## Contents

- mplot3d tutorial
  - Getting started
  - Line plots
  - Scatter plots
  - Wireframe plots
  - Surface plots
  - Tri-Surface plots
  - Contour plots
  - Filled contour plots
  - Polygon plots
  - Bar plots
  - Quiver
  - 2D plots in 3D
  - Text
  - Subplotting

### Getting started

An Axes3D object is created just like any other axes using the projection=’3d’ keyword. Create a new `matplotlib.figure.Figure` and add a new axes to it of type `Axes3D`:

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

New in version 1.0.0: This approach is the preferred method of creating a 3D axes.

**Note:** Prior to version 1.0.0, the method of creating a 3D axes was different. For those using older versions of matplotlib, change `ax = fig.add_subplot(111, projection='3d')` to `ax = Axes3D(fig)`.

### Line plots

`Axes3D.plot(xs, ys, *args, **kwargs)`

Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs, ys</code></td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td><code>zs</code></td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td><code>zdir</code></td>
<td>Which direction to use as z (’x’, ’y’ or ’z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>
Other arguments are passed on to \texttt{plot}()

Scatter plots

\texttt{Axes3D.scatter}(xx, ys, zs=0, zdir='z', s=20, c='b', depthshade=True, *args, **kwargs)

Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{xx}, \texttt{ys}</td>
<td>Positions of data points.</td>
</tr>
<tr>
<td>\texttt{zs}</td>
<td>Either an array of the same length as \texttt{xx} and \texttt{ys} or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>\texttt{zdir}</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>\texttt{s}</td>
<td>size in points$^2$. It is a scalar or an array of the same length as \texttt{x} and \texttt{y}.</td>
</tr>
<tr>
<td>\texttt{c}</td>
<td>a color. \texttt{c} can be a single color format string, or a sequence of color specifications of length \texttt{N}, or a sequence of \texttt{N} numbers to be mapped to colors using the \texttt{cmap} and \texttt{norm} specified via \texttt{kwargs} (see below). Note that \texttt{c} should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. \texttt{c} can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
<tr>
<td>\texttt{depthshade}</td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is True.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to \texttt{scatter}().

Returns a \texttt{Patch3DCollection}.
Wireframe plots

Axes3D.plot_wireframe(X, Y, Z, *args, **kwargs)

Plot a 3D wireframe.

The rstride and cstride kwarg sets the stride used to sample the input data to generate the graph.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size), defaults to 1</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size), defaults to 1</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to LineCollection.

Returns a Line3DCollection
Surface plots

```
Axes3D.plot_surface(X, Y, Z, *args, **kwargs)
```

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the `cmap` argument.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X, Y, Z</code></td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td><code>rstride</code></td>
<td>Array row stride (step size), defaults to 10</td>
</tr>
<tr>
<td><code>cstride</code></td>
<td>Array column stride (step size), defaults to 10</td>
</tr>
<tr>
<td><code>color</code></td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td><code>facecolors</code></td>
<td>Face colors for the individual patches.</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td><code>vmin</code></td>
<td>Minimum value to map</td>
</tr>
<tr>
<td><code>vmax</code></td>
<td>Maximum value to map</td>
</tr>
<tr>
<td><code>shade</code></td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

Other arguments are passed on to `Poly3DCollection`
Tri-Surface plots

**Axes3D.plot_trisurf(***args, **kwargs**)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)  # new in 1.3.1
plot_trisurf(X, Y, triangles=triangles, ...)  # new in 1.3.1
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.
The remaining arguments are:

```
plot_trisurf(..., Z)
```

where \( Z \) is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to `Poly3DCollection`

**Examples:**
New in version 1.2.0: This plotting function was added for the v1.2.0 release.
Contour plots

Axes3D.contour(X, Y, Z, *args, **kwargs)

Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to contour()

Returns a contour
Filled contour plots

```
Axes3D.contourf(X, Y, Z, *args, **kwargs)
```

Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`

Changed in version 1.1.0: The `zdir` and `offset` kwargs were added.
New in version 1.1.0: The feature demoed in the second contourf3d example was enabled as a result of a bugfix for version 1.1.0.

**Polygon plots**

Axes3D.add_collection3d(col, zs=0, zdir='z')

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

**Supported are:**

- PolyCollection
- LineCollection
- PatchCollection
Bar plots

Axes3D.bar(left, height, zs=0, zdir='z', *args, **kwargs)

Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td>height</td>
<td>The height of the bars.</td>
</tr>
<tr>
<td>zs</td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto bar().

Returns a Patch3DCollection
Quiver

Axes3D.quiver(*args, **kwargs)

Plot a 3D field of arrows.

call signatures:

quiver(X, Y, Z, U, V, W, **kwargs)

Arguments:

X, Y, Z: The x, y and z coordinates of the arrow locations

U, V, W: The direction vector that the arrow is pointing

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

length: [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes

arrow_length_ratio: [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3
Any additional keyword arguments are delegated to `LineCollection`
2D plots in 3D

Text

Axes3D.text(x, y, z, zs=0, zdir=z, **kwargs)

Add text to the plot. kwargs will be passed on to Axes.text, except for the zdir keyword, which sets the direction to be used as the z direction.
Subplotting  

Having multiple 3D plots in a single figure is the same as it is for 2D plots. Also, you can have both 2D and 3D plots in the same figure.

New in version 1.0.0: Subplotting 3D plots was added in v1.0.0. Earlier version can not do this.
A tale of 2 subplots

mplot3d API
axes3d

Note: Significant effort went into bringing axes3d to feature-parity with regular axes objects for version 1.1.0. However, more work remains. Please report any functions that do not behave as expected as a bug. In addition, help and patches would be greatly appreciated!

Module containing Axes3D, an object which can plot 3D objects on a 2D matplotlib figure.

class mpl_toolkits.mplot3d.axes3d.Axes3D(fig, rect=None, *args, **kwargs)
Bases: matplotlib.axes._axes.Axes

3D axes object.

add_collection3d(col, zs=0, zdir='z')
Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:
- PolyCollection
- LineCollection
- PatchCollection

add_contour_set(cset, extend3d=False, stride=5, zdir='z', offset=None)

add_contourf_set(cset, zdir='z', offset=None)

auto_scale_xyz(X, Y, Z=None, had_data=None)

autoscale(enable=True, axis='both', tight=None)
Convenience method for simple axis view autoscaling. See matplotlib.axes.Axes.autoscale() for full explanation. Note that this function behaves the same, but for all three axes. Therefore, ‘z’ can be passed for axis, and ‘both’ applies to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

autoscale_view(tight=None, scalex=True, scaley=True, scalez=True)
Autoscale the view limits using the data limits. See matplotlib.axes.Axes.autoscale_view() for documentation. Note that this function applies to the 3D axes, and as such adds the scalez to the function arguments.
Changed in version 1.1.0: Function signature was changed to better match the 2D version. `tight` is now explicitly a kwarg and placed first.

Changed in version 1.2.1: This is now fully functional.

**bar**(*left*, height, zs=0, zdir=’z’, *args, **kwargs*)

Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td>height</td>
<td>The height of the bars.</td>
</tr>
<tr>
<td>zs</td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (’x’, ’y’ or ’z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

**bar3d**(*x*, y, z, dx, dy, dz, color=’b’, zsort=’average’, *args, **kwargs*)

Generate a 3D bar, or multiple bars.

When generating multiple bars, x, y, z have to be arrays. dx, dy, dz can be arrays or scalars.

*color* can be:

- A single color value, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

Keyword arguments are passed onto `Poly3DCollection()`

**can_pan**()

Return `True` if this axes supports the pan/zoom button functionality.

3D axes objects do not use the pan/zoom button.

**can_zoom**()

Return `True` if this axes supports the zoom box button functionality.

3D axes objects do not use the zoom box button.

**cla**()

Clear axes

**clabel**(*args, **kwargs*)

This function is currently not implemented for 3D axes. Returns `None`.  

31.1. mplot3d
**contour**(*X, Y, Z, *args, **kwargs*)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to *contour()*

Returns a *contour*

**contour3D**(*X, Y, Z, *args, **kwargs*)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to *contour()*

Returns a *contour*

**contourf**(*X, Y, Z, *args, **kwargs*)
Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to *contourf()*

Returns a *contourf*

Changed in version 1.1.0: The *zdir* and *offset* kwargs were added.

**contourf3D**(*X, Y, Z, *args, **kwargs*)
Create a 3D contourf plot.

**contour**(*X, Y, Z, *args, **kwargs*)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to *contour()*

Returns a *contour*
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`

Changed in version 1.1.0: The `zdir` and `offset` kwargs were added.

`convert_zunits(z)`
For artists in an axes, if the zaxis has units support, convert z using zaxis unit type
New in version 1.2.1.

`disable_mouse_rotation()`
Disable mouse button callbacks.

`draw(renderer)`

`format_coord(xd, yd)`
Given the 2D view coordinates attempt to guess a 3D coordinate. Looks for the nearest edge to the point and then assumes that the point is at the same z location as the nearest point on the edge.

`format_zdata(z)`
Return z string formatted. This function will use the `fmt_zdata` attribute if it is callable, else will fall back on the zaxis major formatter

`get_autoscale_on()`
Get whether autoscaling is applied for all axes on plot commands
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`get_autoscalez_on()`
Get whether autoscaling for the z-axis is applied on plot commands
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`get_axis_position()`

`get_axisbelow()`
Get whether axis below is true or not.
For axes3d objects, this will always be `True`
New in version 1.1.0: This function was added for completeness.

`get_children()`
get_frame_on()
Get whether the 3D axes panels are drawn
New in version 1.1.0.

get_proj()
Create the projection matrix from the current viewing position.
elev stores the elevation angle in the z plane azim stores the azimuth angle in the x,y plane
dist is the distance of the eye viewing point from the object point.

get_w_lims()
Get 3D world limits.

get_xlim()
Get the x-axis range \([\text{left}, \text{right}]\)
Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_xlim3d()
Get the x-axis range \([\text{left}, \text{right}]\)
Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_ylim()
Get the y-axis range \([\text{bottom}, \text{top}]\)
Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_ylim3d()
Get the y-axis range \([\text{bottom}, \text{top}]\)
Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_zbound()
Returns the z-axis numerical bounds where:
\[\text{lowerBound} < \text{upperBound}\]
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlabel()
Get the z-label text string.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlim()
Get 3D z limits.

get_zlim3d()
Get 3D z limits.

get_zmajorticklabels()
Get the ztick labels as a list of Text instances
New in version 1.1.0.
get_zminorticklabels()

Get the ztick labels as a list of Text instances

Note: Minor ticks are not supported. This function was added only for completeness.

New in version 1.1.0.

get_zscale()

get_zticklabels(minor=False)

Get ztick labels as a list of Text instances. See matplotlib.axes.Axes.get_yticklabels() for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

get_zticklines()

Get ztick lines as a list of Line2D instances. Note that this function is provided merely for completeness. These lines are re-calculated as the display changes.

New in version 1.1.0.

get_zticks(minor=False)

Return the z ticks as a list of locations See matplotlib.axes.Axes.get_yticks() for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

grid(b=True, **kwargs)

Set / unset 3D grid.

Note: Currently, this function does not behave the same as matplotlib.axes.Axes.grid(), but it is intended to eventually support that behavior.

Changed in version 1.1.0: This function was changed, but not tested. Please report any bugs.

have_units()

Return True if units are set on the x, y, or z axes

invert_zaxis()

Invert the z-axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

locator_params(axis='both', tight=None, **kwargs)

Convenience method for controlling tick locators.

See matplotlib.axes.Axes.locator_params() for full documentation Note that this is for
Axes3D objects, therefore, setting `axis` to ‘both’ will result in the parameters being set for all three axes. Also, `axis` can also take a value of ‘z’ to apply parameters to the z axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```python
margins(*args, **kw)
```
Convenience method to set or retrieve autoscaling margins.

**signatures:**

- `margins()`
  returns xmargin, ymargin, zmargin
- `margins(margin)`
- `margins(xmargin, ymargin, zmargin)`
- `margins(x=xmargin, y=ymargin, z=zmargin)`
- `margins(..., tight=False)`

All forms above set the xmargin, ymargin and zmargin parameters. All keyword parameters are optional. A single argument specifies xmargin, ymargin and zmargin. The `tight` parameter is passed to `autoscale_view()`, which is executed after a margin is changed; the default here is `True`, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting `tight` to `None` will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if `xmargin` is not None, then `xmargin` times the X data interval will be added to each end of that interval before it is used in autoscaling.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```python
mouse_init(rotate_btn=1, zoom_btn=3)
```
Initializes mouse button callbacks to enable 3D rotation of the axes. Also optionally sets the mouse buttons for 3D rotation and zooming.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rotate_btn</code></td>
<td>The integer or list of integers specifying which mouse button or buttons to use for 3D rotation of the axes. Default = 1.</td>
</tr>
<tr>
<td><code>zoom_btn</code></td>
<td>The integer or list of integers specifying which mouse button or buttons to use to zoom the 3D axes. Default = 3.</td>
</tr>
</tbody>
</table>

```python
name = u‘3d’
```

```python
plot(xs, ys, *args, **kwargs)
```
Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs, ys</code></td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td><code>zs</code></td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td><code>zdir</code></td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to `plot()`
**plot3D**(*xs, ys, *args, **kwargs*)

Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to **plot**

**plot_surface**(*X, Y, Z, *args, **kwargs*)

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the **cmap** argument.

The **rstride** and **cstride** kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size), defaults to 10</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size), defaults to 10</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>facecolors</td>
<td>Face colors for the individual patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

Other arguments are passed on to **Poly3DCollection**

**plot_trisurf**(*args, **kwargs*)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

The (optional) triangulation can be specified in one of two ways; either:

**plot_trisurf**(triangulation, ...)

where triangulation is a **Triangulation** object, or:
plot_trisurf(X, Y, ...)  
plot_trisurf(X, Y, triangles, ...)  
plot_trisurf(X, Y, triangles=triangles, ...)  

in which case a Triangulation object will be created. See Triangulation for an explanation of these possibilities.

The remaining arguments are:

plot_trisurf(..., Z)  

where Z is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to Poly3DCollection

Examples:
31.1. mplot3d
New in version 1.2.0: This plotting function was added for the v1.2.0 release.

`plot_wireframe(X, Y, Z, *args, **kwargs)`

Plot a 3D wireframe.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X, Y, Z</code></td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td><code>rstride</code></td>
<td>Array row stride (step size), defaults to 1</td>
</tr>
<tr>
<td><code>cstride</code></td>
<td>Array column stride (step size), defaults to 1</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `LineCollection`.

Returns a `Line3DCollection`.

`quiver(*args, **kwargs)`

Plot a 3D field of arrows.

call signatures:

`quiver(X, Y, Z, U, V, W, **kwargs)`

Arguments:

- `X, Y, Z`: The x, y and z coordinates of the arrow locations
The arguments could be array-like or scalars, so long as they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

- **length**: [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes
- **arrow_length_ratio**: [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3

Any additional keyword arguments are delegated to LineCollection

```python
quiver3D(*args, **kwargs)
```

Plot a 3D field of arrows.

call signatures:

```python
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

- **X**, **Y**, **Z**: The x, y and z coordinates of the arrow locations
- **U**, **V**, **W**: The direction vector that the arrow is pointing

The arguments could be array-like or scalars, so long as they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

- **length**: [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes
- **arrow_length_ratio**: [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3

Any additional keyword arguments are delegated to LineCollection

```python
scatter(xs, ys, zs=0, zdir=u'z', s=20, c=u'b', depthshade=True, *args, **kwargs)
```

Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>Positions of data points.</td>
</tr>
<tr>
<td>zs</td>
<td>Either an array of the same length as xs and ys or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>s</td>
<td>Size in points^2. It is a scalar or an array of the same length as x and y.</td>
</tr>
<tr>
<td>c</td>
<td>A color. c can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). Note that c should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. c can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
<tr>
<td>depthshade</td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is True.</td>
</tr>
</tbody>
</table>
Keyword arguments are passed on to `scatter()`.

Returns a `Patch3DCollection`

```python
scatter3D(xs, ys, zs=0, zdir='z', s=20, c='b', depthshade=True, *args, **kwargs)
```

Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs, ys</code></td>
<td>Positions of data points.</td>
</tr>
<tr>
<td><code>zs</code></td>
<td>Either an array of the same length as <code>xs</code> and <code>ys</code> or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td><code>zdir</code></td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td><code>s</code></td>
<td>size in points^2. It is a scalar or an array of the same length as <code>x</code> and <code>y</code>.</td>
</tr>
<tr>
<td><code>c</code></td>
<td>a color. <code>c</code> can be a single color format string, or a sequence of color specifications of length <code>N</code>, or a sequence of <code>N</code> numbers to be mapped to colors using the <code>cmap</code> and <code>norm</code> specified via kwargs (see below). Note that <code>c</code> should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. <code>c</code> can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
<tr>
<td><code>depthshade</code></td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is <code>True</code>.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `scatter()`.

Returns a `Patch3DCollection`

```python
set_autoscale_on(b)
```

Set whether autoscaling is applied on plot commands

accepts: [ `True` | `False` ]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```python
set_autoscalez_on(b)
```

Set whether autoscaling for the z-axis is applied on plot commands

accepts: [ `True` | `False` ]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```python
set_axis_off()
```

```python
set_axis_on()
```

```python
set_axisbelow(b)
```

Set whether the axis ticks and gridlines are above or below most artists

For axes3d objects, this will ignore any settings and just use `True`

accepts: [ `True` | `False` ]

New in version 1.1.0: This function was added for completeness.

```python
set_frame_on(b)
```

Set whether the 3D axes panels are drawn
set_title(label, fontdict=None, loc='center', **kwargs)

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

- **label**: str
  - Text to use for the title
- **fontdict**: dict
  - A dictionary controlling the appearance of the title text, the default fontdict is:
    ```
    {'fontsize': rcParams['axes.titlesize'],
     'fontweight' : rcParams['axes.titleweight'],
     'verticalalignment': 'baseline',
     'horizontalalignment': loc}
    ```
- **loc**: {'center', 'left', 'right'}, str, optional
  - Which title to set, defaults to 'center'

**Returns**

- **text**: Text
  - The matplotlib text instance representing the title

**Other Parameters**

- **kwargs**: text properties
  - Other keyword arguments are text properties, see Text for a list of valid text properties.

set_xlim(left=None, right=None, emit=True, auto=False, **kw)

Set 3D x limits.

See matplotlib.axes.Axes.set_xlim() for full documentation.

set_xlim3d(left=None, right=None, emit=True, auto=False, **kw)

Set 3D x limits.

See matplotlib.axes.Axes.set_xlim() for full documentation.

set_xscale(value, **kwars)

Call signature:

```
set_xscale(value)
```

Set the scaling of the x-axis: u'linear' | u'log' | u'symlog'

**ACCEPTS**: [u'linear' | u'log' | u'symlog']

**Different kwars are accepted, depending on the scale**: ‘linear’

- **‘log**
  - **baseX/baseY**: The base of the logarithm
  - **nonposx/nonposy**: ['mask' | 'clip'] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
**subsx/subsy**: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘symlog’

**basex/basey**: The base of the logarithm

**linthreshx/linthreshy**: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

**subsx/subsy**: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

**linscalex/linscaley**: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when linscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_ylim**(bottom=None, top=None, emit=True, auto=False, **kw)

Set 3D y limits.

See matplotlib.axes.Axes.set_ylim() for full documentation.

**set_ylim3d**(bottom=None, top=None, emit=True, auto=False, **kw)

Set 3D y limits.

See matplotlib.axes.Axes.set_ylim() for full documentation.

**set_yscale**(value, **kwargs)

Call signature:

```
set_yscale(value)
```

Set the scaling of the y-axis: ‘linear’ | ‘log’ | ‘symlog’

ACCEPTS: [‘linear’ | ‘log’ | ‘symlog’]

**Different kwargs are accepted, depending on the scale**: ‘linear’

‘log’

**basex/basey**: The base of the logarithm

**nonposx/nonposy**: [‘mask’ | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

**subsx/subsy**: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘symlog’

**basex/basey**: The base of the logarithm

**linthreshx/linthreshy**: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

linscalex/linscaley: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when linscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zbound(lower=None, upper=None)
Set the lower and upper numerical bounds of the z-axis. This method will honor axes inversion regardless of parameter order. It will not change the _autoscaleZon attribute.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zlabel(zlabel, fontdict=None, labelpad=None, **kwargs)
Set zlabel. See doc for set_ylabel() for description.

Note: Currently, labelpad does not have an effect on the labels.

set_zlim(bottom=None, top=None, emit=True, auto=False, **kw)
Set 3D z limits.

See matplotlib.axes.Axes.set_xlim() for full documentation

set_zlim3d(bottom=None, top=None, emit=True, auto=False, **kw)
Set 3D z limits.

See matplotlib.axes.Axes.set_xlim() for full documentation

set_zmargin(m)
Set padding of Z data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zscale(value, **kwargs)
call signature:

set_zscale(value)

Set the scaling of the z-axis: u'linear' | u'log' | u'symlog'

ACCEPTS: [u'linear' | u'log' | u'symlog']

Different kwargs are accepted, depending on the scale: 'linear'

'log'

basex/basey: The base of the logarithm
**nonposx/nonposy:** ['mask', 'clip'] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

**subsx/subsy:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

'symlog'

**basex/basey:** The base of the logarithm

**linthreshx/linthreshy:** The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

**subsx/subsy:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

**linscalex/linscaley:** This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when linscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

---

**Note:** Currently, Axes3D objects only supports linear scales. Other scales may or may not work, and support for these is improving with each release.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zticklabels(**args, **kwargs)**
Set z-axis tick labels. See `matplotlib.axes.Axes.set_yticklabels()` for more details.

**Note:** Minor ticks are not supported by Axes3D objects.

New in version 1.1.0.

**set_zticks(**args, **kwargs)**
Set z-axis tick locations. See `matplotlib.axes.Axes.set_yticks()` for more details.

**Note:** Minor ticks are not supported.

New in version 1.1.0.

**text**(x, y, z, s, zdir=None, **kwargs)
Add text to the plot. kwargs will be passed on to Axes.text, except for the zdir keyword, which sets the direction to be used as the z direction.

**text2D**(x, y, s, fontdict=None, withdash=False, **kwargs)
Add text to the axes.

Add text in string s to axis at location x, y, data coordinates.

**Parameters**

- **x, y:** scalars
data coordinates
- **s:** string
text
**fontdict** : dictionary, optional, default: None

A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

**withdash** : boolean, optional, default: False

Creates a `TextWithDash` instance instead of a `Text` instance.

**Other Parameters** **kwargs** : `Text` properties.

Other miscellaneous text parameters.

Examples

Individual keyword arguments can be used to override any given parameter:

```python
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```python
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
        ... verticalalignment='center',
        ... transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `bbox`. `bbox` is a dictionary of `Rectangle` properties. For example:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

**text3D**(x, y, z, s, zdir=None, **kwargs)

Add text to the plot. `kwargs` will be passed on to Axes.text, except for the `zdir` keyword, which sets the direction to be used as the z direction.

**tick_params**(axis='both', **kwargs)

Convenience method for changing the appearance of ticks and tick labels.

See `matplotlib.axes.Axes.tick_params()` for more complete documentation.

The only difference is that setting `axis` to 'both' will mean that the settings are applied to all three axes. Also, the `axis` parameter also accepts a value of 'z', which would mean to apply to only the z-axis.

Also, because of how Axes3D objects are drawn very differently from regular 2D axes, some of these settings may have ambiguous meaning. For simplicity, the 'z' axis will accept settings as if it was like the 'y' axis.

**Note:** While this function is currently implemented, the core part of the Axes3D object may ignore some of these settings. Future releases will fix this. Priority will be given to those who file bugs.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.
**ticklabel_format(**kwargs)**

Convenience method for manipulating the ScalarFormatter used by default for linear axes in
Axed3D objects.

See `matplotlib.axes.Axes.ticklabel_format()` for full documentation. Note that this version applies to all three axes of the Axes3D object. Therefore, the axis argument will also accept a value of ‘z’ and the value of ‘both’ will apply to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**tricontour(**args, **kwargs)**

Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

**tricontourf(**args, **kwargs)**

Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

**tunit_cube**(vals=None, M=None)
**tunit_edges** *(vals=None, M=None)*

**unit_cube** *(vals=None)*

**update_datalim** *(xys, **kwargs)*

**view_init** *(elev=None, azim=None)*

Set the elevation and azimuth of the axes.

This can be used to rotate the axes programatically.

‘elev’ stores the elevation angle in the z plane. ‘azim’ stores the azimuth angle in the x,y plane.

if elev or azim are None (default), then the initial value is used which was specified in the Axes3D constructor.

**zaxis_date** *(tz=None)*

Sets up z-axis ticks and labels that treat the z data as dates.

*tz* is a timezone string or tzinfo instance. Defaults to rc value.

**zaxis_inverted**()

Returns True if the z-axis is inverted.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**mpl_toolkits.mplot3d.axes3d.get_test_data** *(delta=0.05)*

Return a tuple X, Y, Z with a test data set.

**mpl_toolkits.mplot3d.axes3d.unit_bbox**()

**axis3d**

Note: Historically, axis3d has suffered from having hard-coded constants controlling the look and feel of the 3D plot. This precluded user level adjustments such as label spacing, font colors and panel colors. For version 1.1.0, these constants have been consolidated into a single private member dictionary, self._axinfo, for the axis object. This is intended only as a stop-gap measure to allow user-level customization, but it is not intended to be permanent.

**class mpl_toolkits.mplot3d.axis3d.Axis** *(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)*

Bases: matplotlib.axis.XAxis

**draw** *(renderer)*
draw_pane(renderer)

get_major_ticks(numticks=None)

get_rotate_label(text)

get_tick_positions()

get_tightbbox(renderer)

get_view_interval()
    return the Interval instance for this 3d axis view limits
init3d()

set_pane_color(color)
    Set pane color to a RGBA tuple
set_pane_pos(xys)

set_rotate_label(val)
    Whether to rotate the axis label: True, False or None. If set to None the label will be rotated if
longer than 4 chars.
set_view_interval(vmin, vmax, ignore=False)

class mpl_toolkits.mplot3d.axis3d.XAxis(adir, v_intervalx, d_intervalx, axes, *args,
    **kwargs)
    Bases: mpl_toolkits.mplot3d.axis3d.Axis
get_data_interval()
    return the Interval instance for this axis data limits

class mpl_toolkits.mplot3d.axis3d.YAxis(adir, v_intervalx, d_intervalx, axes, *args,
    **kwargs)
    Bases: mpl_toolkits.mplot3d.axis3d.Axis
get_data_interval()
    return the Interval instance for this axis data limits

class mpl_toolkits.mplot3d.axis3d.ZAxis(adir, v_intervalx, d_intervalx, axes, *args,
    **kwargs)
    Bases: mpl_toolkits.mplot3d.axis3d.Axis
get_data_interval()
    return the Interval instance for this axis data limits
mpl_toolkits.mplot3d.axis3d.get_flip_min_max(coord, index, mins, maxs)
`mpl_toolkits.mplot3d.axis3d.move_from_center(coord, centers, deltas, axmask=(True, True, True))`

Return a coordinate that is moved by “deltas” away from the center.

`mpl_toolkits.mplot3d.axis3d.tick_update_position(tick, tickxs, tickys, labelpos)`

Update tick line and label position and style.

**art3d** Module containing 3D artist code and functions to convert 2D artists into 3D versions which can be added to an Axes3D.

**class mpl_toolkits.mplot3d.art3d.Line3D(xs, ys, zs, *args, **kwargs)**

Bases: `matplotlib.lines.Line2D`

3D line object.

Keyword arguments are passed onto `Line2D()`.

`draw(renderer)`

`set_3d_properties(zs=0, zdir=u'z')`

**class mpl_toolkits.mplot3d.art3d.Line3DCollection(segments, *args, **kwargs)**

Bases: `matplotlib.collections.LineCollection`

A collection of 3D lines.

Keyword arguments are passed onto `LineCollection()`.

`do_3d_projection(renderer)`

Project the points according to renderer matrix.

`draw(renderer, project=False)`

`set_segments(segments)`

Set 3D segments

`set_sort_zpos(val)`

Set the position to use for z-sorting.

**class mpl_toolkits.mplot3d.art3d.Patch3D(*args, **kwargs)**

Bases: `matplotlib.patches.Patch`

3D patch object.

`do_3d_projection(renderer)`

`draw(renderer)`

`get_facecolor()`
get_path()

set_3d_properties(verts, zs=0, zdir='z')

class mpl_toolkits.mplot3d.art3d.Patch3DCollection(*args, **kwargs)
   Bases: matplotlib.collections.PatchCollection

   A collection of 3D patches.
   Create a collection of flat 3D patches with its normal vector pointed in zdir direction, and located at zs on the zdir axis. ‘zs’ can be a scalar or an array-like of the same length as the number of patches in the collection.
   Constructor arguments are the same as for PatchCollection. In addition, keywords zs=0 and zdir='z' are available.

   Also, the keyword argument “depthshade” is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is True). This is typically desired in scatter plots.

do_3d_projection(renderer)

set_3d_properties(zs, zdir)

set_sort_zpos(val)
    Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.Path3DCollection(*args, **kwargs)
   Bases: matplotlib.collections.PathCollection

   A collection of 3D paths.
   Create a collection of flat 3D paths with its normal vector pointed in zdir direction, and located at zs on the zdir axis. ‘zs’ can be a scalar or an array-like of the same length as the number of paths in the collection.
   Constructor arguments are the same as for PathCollection. In addition, keywords zs=0 and zdir='z' are available.

   Also, the keyword argument “depthshade” is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is True). This is typically desired in scatter plots.

do_3d_projection(renderer)

set_3d_properties(zs, zdir)

set_sort_zpos(val)
    Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.PathPatch3D(path, **kwargs)
   Bases: mpl_toolkits.mplot3d.art3d.Patch3D
3D PathPatch object.

**do_3d_projection**(renderer)

**set_3d_properties**(path, zs=0, zdir='z')

```
class mpl_toolkits.mplot3d.art3d.Poly3DCollection(verts, *args, **kwargs)

Bases: matplotlib.collections.PolyCollection
```

A collection of 3D polygons.

Create a Poly3DCollection.

verts should contain 3D coordinates.

Keyword arguments: zs, see set_zsort for options.

Note that this class does a bit of magic with the _facecolors and _edgecolors properties.

**do_3d_projection**(renderer)

Perform the 3D projection for this object.

**draw**(renderer)

**get_edgecolor**()

**get_edgecolors**()

**get_facecolor**()

**get_facecolors**()

**get_vector**(segments3d)

Optimize points for projection

**set_3d_properties**()

**set_alpha**(alpha)

Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

**set_edgecolor**(colors)

**set_edgecolors**(colors)

**set_facecolor**(colors)
set_facecolors(colors)

set_sort_zpos(val)
   Set the position to use for z-sorting.

set_verts(verts, closed=True)
   Set 3D vertices.

set_zsort(zsort)

   Set z-sorting behaviour: boolean: if True use default ‘average’ string: ‘average’, ‘min’ or ‘max’

class mpl_toolkits.mplot3d.art3d.Text3D(x=0, y=0, z=0, text=u'', zdir=u'z', **kwargs)
   Bases: matplotlib.text.Text

   Text object with 3D position and (in the future) direction.
   x, y, z Position of text text Text string to display zdir Direction of text
   Keyword arguments are passed onto Text().

draw(renderer)

set_3d_properties(z=0, zdir=u'z')

mpl_toolkits.mplot3d.art3d.get_colors(c, num)
   Stretch the color argument to provide the required number num

mpl_toolkits.mplot3d.art3d.get_dir_vector(zdir)

mpl_toolkits.mplot3d.art3d.get_patch_verts(patch)
   Return a list of vertices for the path of a patch.

mpl_toolkits.mplot3d.art3d.iscolor(c)

mpl_toolkits.mplot3d.art3d.juggle_axes(xs, ys, zs, zdir)
   Reorder coordinates so that 2D xs, ys can be plotted in the plane orthogonal to zdir. zdir is normally x,
   y or z. However, if zdir starts with a ‘-‘ it is interpreted as a compensation for rotate_axes.

mpl_toolkits.mplot3d.art3d.line_2d_to_3d(line, zs=0, zdir=u'z')
   Convert a 2D line to 3D.

mpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d(col, zs=0, zdir=u'z')
   Convert a LineCollection to a Line3DCollection object.

mpl_toolkits.mplot3d.art3d.norm_angle(a)
   Return angle between -180 and +180

mpl_toolkits.mplot3d.art3d.norm_text_angle(a)
   Return angle between -90 and +90
mpl_toolkits.mplot3d.art3d.patch_2d_to_3d(patch, z=0, zdir=u’z’)
Convert a Patch to a Patch3D object.

mpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d(col, zs=0, zdir=u’z’,
depthshade=True)
Convert a PatchCollection into a Patch3DCollection object (or a PathCollection into a Path3DCollection object).

Keywords:
z The location or locations to place the patches in the collection along the zdir axis. Defaults to 0.
zdir The axis in which to place the patches. Default is “z”.
depthshade Whether to shade the patches to give a sense of depth. Defaults to True.

mpl_toolkits.mplot3d.art3d.path_to_3d_segment(path, zs=0, zdir=u’z’)
Convert a path to a 3D segment.

mpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d(pathpatch, z=0, zdir=u’z’)
Convert a PathPatch to a PathPatch3D object.

mpl_toolkits.mplot3d.art3d.paths_to_3d_segments(paths, zs=0, zdir=u’z’)
Convert paths from a collection object to 3D segments.

mpl_toolkits.mplot3d.art3d.poly_collection_2d_to_3d(col, zs=0, zdir=u’z’)
Convert a PolyCollection to a Poly3DCollection object.

mpl_toolkits.mplot3d.art3d.rotate_axes(xs, ys, zs, zdir)
Reorder coordinates so that the axes are rotated with zdir along the original z axis. Prepending the axis with a ‘-’ does the inverse transform, so zdir can be x, -x, y, -y, z or -z.

mpl_toolkits.mplot3d.art3d.text_2d_to_3d(obj, zs=0, zdir=u’z’)
Convert a Text to a Text3D object.

mpl_toolkits.mplot3d.art3d.zalpha(colors, zs)
Modify the alphas of the color list according to depth.

proj3d Various transforms used for by the 3D code

mpl_toolkits.mplot3d.proj3d.inv_transform(xs, ys, zs, M)

mpl_toolkits.mplot3d.proj3d.line2d(p0, p1)
Return 2D equation of line in the form ax+by+c = 0

mpl_toolkits.mplot3d.proj3d.line2d_dist(l, p)
Distance from line to point line is a tuple of coefficients a,b,c

mpl_toolkits.mplot3d.proj3d.line2d_seg_dist(p1, p2, p0)
distance(s) from line defined by p1 - p2 to point(s) p0
p0[0] = x(s) p0[1] = y(s)
intersection point p = p1 + u*(p2-p1) and intersection point lies within segment if u is between 0 and 1

mpl_toolkits.mplot3d.proj3d.mod(v)
3d vector length
mpl_toolkits.mplot3d.proj3d.persp_transformation(zfront, zback)

mpl_toolkits.mplot3d.proj3d.proj_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_trans_clip_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_trans_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_transform(xs, ys, zs, M)
    Transform the points by the projection matrix

mpl_toolkits.mplot3d.proj3d.proj_transform_clip(xs, ys, zs, M)
    Transform the points by the projection matrix and return the clipping result returns txs,tsy,tzs,tis

mpl_toolkits.mplot3d.proj3d.proj_transform_vec(vec, M)

mpl_toolkits.mplot3d.proj3d.proj_transform_vec_clip(vec, M)

mpl_toolkits.mplot3d.proj3d.rot_x(V, alpha)

mpl_toolkits.mplot3d.proj3d.test_lines_dists()

mpl_toolkits.mplot3d.proj3d.test_proj()

mpl_toolkits.mplot3d.proj3d.test_proj_draw_axes(M, s=1)

mpl_toolkits.mplot3d.proj3d.test_proj_make_M(E=None)

mpl_toolkits.mplot3d.proj3d.test_rot()

mpl_toolkits.mplot3d.proj3d.test_world()

mpl_toolkits.mplot3d.proj3d.transform(xs, ys, zs, M)
    Transform the points by the projection matrix

mpl_toolkits.mplot3d.proj3d.vec_pad_ones(xs, ys, zs)

mpl_toolkits.mplot3d.proj3d.view_transformation(E, R, V)

mpl_toolkits.mplot3d.proj3d.world_transformation(xmin, xmax, ymin, ymax, zmin, zmax)
mplot3d FAQ

How is mplot3d different from MayaVi?  MayaVi2 is a very powerful and featureful 3D graphing library. For advanced 3D scenes and excellent rendering capabilities, it is highly recommended to use MayaVi2.

mplot3d was intended to allow users to create simple 3D graphs with the same “look-and-feel” as matplotlib’s 2D plots. Furthermore, users can use the same toolkit that they are already familiar with to generate both their 2D and 3D plots.

My 3D plot doesn’t look right at certain viewing angles  This is probably the most commonly reported issue with mplot3d. The problem is that – from some viewing angles – a 3D object would appear in front of another object, even though it is physically behind it. This can result in plots that do not look “physically correct.”

Unfortunately, while some work is being done to reduce the occurrence of this artifact, it is currently an intractable problem, and cannot be fully solved until matplotlib supports 3D graphics rendering at its core.

The problem occurs due to the reduction of 3D data down to 2D + z-order scalar. A single value represents the 3rd dimension for all parts of 3D objects in a collection. Therefore, when the bounding boxes of two collections intersect, it becomes possible for this artifact to occur. Furthermore, the intersection of two 3D objects (such as polygons or patches) can not be rendered properly in matplotlib’s 2D rendering engine.

This problem will likely not be solved until OpenGL support is added to all of the backends (patches are greatly welcomed). Until then, if you need complex 3D scenes, we recommend using MayaVi.

I don’t like how the 3D plot is laid out, how do I change that?  Historically, mplot3d has suffered from a hard-coding of parameters used to control visuals such as label spacing, tick length, and grid line width. Work is being done to eliminate this issue. For matplotlib v1.1.0, there is a semi-official manner to modify these parameters. See the note in the axis3d section of the mplot3d API documentation for more information.

mpl_toolkits.mplot3d provides some basic 3D plotting (scatter, surf, line, mesh) tools. Not the fastest or feature complete 3D library out there, but ships with matplotlib and thus may be a lighter weight solution for some use cases.
31.2 AxesGrid

31.2.1 Matplotlib AxesGrid Toolkit

The matplotlib AxesGrid toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. While the aspect parameter in matplotlib adjust the position of the single axes, AxesGrid toolkit provides a framework to adjust the position of multiple axes according to their aspects.
Matplotlib, Release 1.4.3

Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of `axes_grid`. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (`axes_grid1` and `axisartist`). While `axes_grid` namespace is maintained for the backward compatibility, use of `axes_grid1` and `axisartist` is recommended.

Warning: `axes_grid` and `axisartist` (but not `axes_grid1`) uses a custom Axes class (derived from the mpl's original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use `axes_grid1` to avoid this, or see how things are different in `axes_grid` and `axisartist` (LINK needed)

Overview of AxesGrid toolkit

What is AxesGrid toolkit?

The matplotlib AxesGrid toolkit is a collection of helper classes, mainly to ease displaying (multiple) images in matplotlib.

Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of `axes_grid`. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (`axes_grid1` and `axisartist`). While `axes_grid` namespace is maintained for the backward compatibility, use of `axes_grid1` and `axisartist` is recommended.

Warning: `axes_grid` and `axisartist` (but not `axes_grid1`) uses a custom Axes class (derived from the mpl's original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use `axes_grid1` to avoid this, or see how things are different in `axes_grid` and `axisartist` (LINK needed)

AxesGrid toolkit has two namespaces (`axes_grid1` and `axisartist`). `axisartist` contains custom Axes class that is meant to support for curvilinear grids (e.g., the world coordinate system in astronomy). Unlike mpl's original Axes class which uses Axes.xaxis and Axes.yaxis to draw ticks, ticklines and etc., Axes in axisartist uses special artist (AxisArtist) which can handle tick, ticklines and etc. for curved coordinate systems.
Since it uses a special artists, some mpl commands that work on Axes.xaxis and Axes.yaxis may not work. See LINK for more detail.

`axes_grid1` is a collection of helper classes to ease displaying (multiple) images with matplotlib. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates, which may not be ideal for displaying images that needs to have a given aspect ratio. For example, it helps you to have a colorbar whose height always matches that of the image. `ImageGrid`, `RGB Axes` and `AxesDivider` are helper classes that deals with adjusting the location of (multiple) Axes. They provides a framework to adjust the position of multiple axes at the drawing time. `ParasiteAxes` provides twinx(or twiny)-like features so that you can plot different data (e.g., different y-scale) in a same Axes. `AnchoredArtists` includes custom artists which are placed at some anchored position, like the legend.
AXES_GRID1

ImageGrid  A class that creates a grid of Axes. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio. For example, displaying images of a same size with some fixed padding between them cannot be easily done in matplotlib. ImageGrid is used in such case.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
nrows_ncols = (2, 2),  # creates 2x2 grid of axes
axes_pad=0.1,  # pad between axes in inch.
)

for i in range(4):
    grid[i].imshow(im)  # The AxesGrid object work as a list of axes.

plt.show()
```

- The position of each axes is determined at the drawing time (see AxesDivider), so that the size of the entire grid fits in the given rectangle (like the aspect of axes). Note that in this example, the paddings between axes are fixed even if you changes the figure size.
Matplotlib, Release 1.4.3

- axes in the same column has a same axes width (in figure coordinate), and similarly, axes in the same row has a same height. The widths (height) of the axes in the same row (column) are scaled according to their view limits (xlim or ylim).

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

F = plt.figure(1, (5.5, 3.5))
grid = ImageGrid(F, 111, # similar to subplot(111)
                 nrows_ncols=(1, 3),
                 axes_pad=0.1,
                 add_all=True,
                 label_mode="L",
                 )

Z, extent = get_demo_image() # demo image

im1 = Z
im2 = Z[:,:10]
im3 = Z[:,10:]

vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = grid[i]
    ax.imshow(im, origin="lower", vmin=vmin, vmax=vmax, interpolation="nearest")

plt.draw()
plt.show()
```

646 Chapter 31. General Toolkits
• xaxis are shared among axes in a same column. Similarly, yaxis are shared among axes in a same row. Therefore, changing axis properties (view limits, tick location, etc. either by plot commands or using your mouse in interactive backends) of one axes will affect all other shared axes.

When initialized, ImageGrid creates given number (ngrids or ncols * nrows if ngrids is None) of Axes instances. A sequence-like interface is provided to access the individual Axes instances (e.g., grid[0] is the first Axes in the grid. See below for the order of axes).

AxesGrid takes following arguments,

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figrect</td>
<td></td>
<td>number of rows and cols. e.g., (2,2)</td>
</tr>
<tr>
<td>nrows_ncols</td>
<td>None</td>
<td>number of grids. nrows x ncols if None</td>
</tr>
<tr>
<td>direction</td>
<td>“row”</td>
<td>increasing direction of axes number. [row</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>pad between axes in inches</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>Add axes to figures if True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>xaxis &amp; yaxis of all axes are shared if True</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>aspect of axes</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>location of tick labels thaw will be displayed. “1” (only the lower left axes), “L” (left most and bottom most axes), or “all”.</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>pad between image axes and colorbar axes</td>
</tr>
<tr>
<td>cbar_location</td>
<td>right</td>
<td>size of the colorbar</td>
</tr>
</tbody>
</table>

31.2. AxesGrid

647
**rect** specifies the location of the grid. You can either specify coordinates of the rectangle to be used (e.g., (0.1, 0.1, 0.8, 0.8) as in the Axes), or the subplot-like position (e.g., “121”).

**direction** means the increasing direction of the axes number.

**aspect** By default (False), widths and heights of axes in the grid are scaled independently. If True, they are scaled according to their data limits (similar to aspect parameter in mpl).

**share_all** if True, xaxis and yaxis of all axes are shared.

**direction** direction of increasing axes number. For “row”,

```
grid[0]   grid[1]
```

For “column”,

```
grid[0]   grid[2]
```

You can also create a colorbar (or colorbars). You can have colorbar for each axes (cbar_mode=”each”), or you can have a single colorbar for the grid (cbar_mode=”single”). The colorbar can be placed on your right, or top. The axes for each colorbar is stored as a cbar_axes attribute.

The examples below show what you can do with AxesGrid.

**AxesDivider** Behind the scene, the ImageGrid class and the RGBAxes class utilize the AxesDivider class, whose role is to calculate the location of the axes at drawing time. While a more about the AxesDivider is (will be) explained in (yet to be written) AxesDividerGuide, direct use of the AxesDivider class will not be necessary for most users. The axes_divider module provides a helper function make_axes_locatable, which can be useful. It takes a existing axes instance and create a divider for it.

```
ax = subplot(1,1,1)
divider = make_axes_locatable(ax)
```

`make_axes_locatable` returns an instance of the AxesLocator class, derived from the Locator. It provides `append_axes` method that creates a new axes on the given side of (“top”, “right”, “bottom” and “left”) of the original axes.

**colorbar whose height (or width) in sync with the master axes**

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import numpy as np
```
scatter_hist.py with AxesDivider  The “scatter_hist.py” example in mpl can be rewritten using
make_axes_locatable.

axScatter = subplot(111)
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", size=1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", size=1.2, pad=0.1, sharey=axScatter)

# the scatter plot:
# histograms
bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

See the full source code below.

The scatter_hist using the AxesDivider has some advantage over the original scatter_hist.py in mpl. For example, you can set the aspect ratio of the scatter plot, even with the x-axis or y-axis is shared accordingly.

**ParasiteAxes** The ParasiteAxes is an axes whose location is identical to its host axes. The location is adjusted in the drawing time, thus it works even if the host change its location (e.g., images).

In most cases, you first create a host axes, which provides a few method that can be used to create parasite axes. They are *twinx*, *twiny* (which are similar to twinx and twiny in the matplotlib) and *twin*. *twin* takes an arbitrary transformation that maps between the data coordinates of the host axes and the parasite axes. *draw* method of the parasite axes are never called. Instead, host axes collects artists in parasite axes and draw them as if they belong to the host axes, i.e., artists in parasite axes are merged to those of the host axes and then drawn according to their zorder. The host and parasite axes modifies some of the axes behavior.
For example, color cycle for plot lines are shared between host and parasites. Also, the legend command in host, creates a legend that includes lines in the parasite axes. To create a host axes, you may use `host_subplot` or `host_axes` command.

**Example 1. tinx**

```python
from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt

host = host_subplot(111)
par = host.twinx()

host.set_xlabel("Distance")
host.set_ylabel("Density")
par.set_ylabel("Temperature")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par.plot([0, 1, 2], [0, 3, 2], label="Temperature")

leg = plt.legend()

host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())

par.yaxis.get_label().set_color(p2.get_color())
leg.texts[1].set_color(p2.get_color())

plt.show()
```
Example 2. twin  *twin* without a transform argument treat the parasite axes to have a same data transform as the host. This can be useful when you want the top(or right)-axis to have different tick-locations, tick-labels, or tick-formatter for bottom(or left)-axis.

```python
ax2 = ax.twin()  # now, ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
ax2.set_xticklabels(["0", r"$\frac{1}{2}\pi$", r"$\pi$", r"$\frac{3}{2}\pi$", r"$2\pi$"],
   r"$\pi$", r"$\frac{3}{2}\pi$", r"$2\pi$"))
```
A more sophisticated example using twin. Note that if you change the x-limit in the host axes, the x-limit of the parasite axes will change accordingly.
AnchoredArtists  It’s a collection of artists whose location is anchored to the (axes) bbox, like the legend. It is derived from OffsetBox in mpl, and artist need to be drawn in the canvas coordinate. But, there is a limited support for an arbitrary transform. For example, the ellipse in the example below will have width and height in the data coordinate.

```python
import matplotlib.pyplot as plt
def draw_text(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredText
    at = AnchoredText("Figure 1a",
                      loc=2, prop=dict(size=8), frameon=True,
                      )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
                       loc=3, prop=dict(size=8), frameon=True,
                       bbox_to_anchor=(0., 1.),
                       bbox_transform=ax.transAxes
                      )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at2)
```

AnchoredArtists is a collection of artists whose location is anchored to the (axes) bbox, like the legend. It is derived from OffsetBox in mpl, and artist need to be drawn in the canvas coordinate. But, there is a limited support for an arbitrary transform. For example, the ellipse in the example below will have width and height in the data coordinate.

```python
import matplotlib.pyplot as plt
def draw_text(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredText
    at = AnchoredText("Figure 1a",
                      loc=2, prop=dict(size=8), frameon=True,
                      )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
                       loc=3, prop=dict(size=8), frameon=True,
                       bbox_to_anchor=(0., 1.),
                       bbox_transform=ax.transAxes
                      )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at2)
```
```python
def draw_circle(ax):
    # circle in the canvas coordinate
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                               loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)

def draw_ellipse(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                          loc=3, pad=0.5, borderpad=0.4, frameon=True)
    ax.add_artist(ae)

def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
                           0.1,
                           r"1$'\prime$",
                           loc=8,
                           pad=0.1, borderpad=0.5, sep=5,
                           frameon=False)
    ax.add_artist(asb)

if 1:
    ax = plt.gca()
    ax.set_aspect(1.)
    draw_text(ax)
    draw_circle(ax)
    draw_ellipse(ax)
    draw_sizebar(ax)
    plt.show()
```

31.2. AxesGrid
InsetLocator  mpl_toolkits.axes_grid.inset_locator provides helper classes and functions to place your (inset) axes at the anchored position of the parent axes, similarly to AnchoredArtist.

Using mpl_toolkits.axes_grid.inset_locator.inset_axes(), you can have inset axes whose size is either fixed, or a fixed proportion of the parent axes. For example:

```python
inset_axes = inset_axes(parent_axes,
                        width="30%", # width = 30% of parent_bbox
                        height=1., # height : 1 inch
                        loc=3)
```

creates an inset axes whose width is 30% of the parent axes and whose height is fixed at 1 inch.

You may creates your inset whose size is determined so that the data scale of the inset axes to be that of the parent axes multiplied by some factor. For example,

```python
inset_axes = zoomed_inset_axes(ax,
                                0.5, # zoom = 0.5
                                loc=1)
```

creates an inset axes whose data scale is half of the parent axes. Here is complete examples.
For example, zoomed_inset_axes() can be used when you want the inset represents the zoom-up of the small portion in the parent axes. And mpl_toolkits/axes_grid/inset_locator provides a helper function mark_inset() to mark the location of the area represented by the inset axes.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
import numpy as np

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

fig, ax = plt.subplots(figsize=[5,4])

# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30+ny, 30:30+nx] = Z

# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest",
          origin="lower")

axins = zoomed_inset_axes(ax, 6, loc=1) # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest",
             origin="lower")
```

31.2. AxesGrid
# sub region of the original image
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)

plt.xticks(visible=False)
plt.yticks(visible=False)

# draw a bbox of the region of the inset axes in the parent axes and
# connecting lines between the bbox and the inset axes area
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

plt.draw()
plt.show()

RGB Axes  RGBAxes is a helper class to conveniently show RGB composite images. Like ImageGrid, the location of axes are adjusted so that the area occupied by them fits in a given rectangle. Also, the xaxis and yaxis of each axes are shared.

from mpl_toolkits.axes_grid1.axes_rgb import RGBAxes

fig = plt.figure(1)
ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8])

r, g, b = get_rgb()  # r, g, b are 2-d images
ax.imshow_rgb(r, g, b,
             origin="lower", interpolation="nearest")
AXISARTIST

AxisArtist AxisArtist module provides a custom (and very experimental) Axes class, where each axis (left, right, top and bottom) have a separate artist associated which is responsible to draw axis-line, ticks, ticklabels, label. Also, you can create your own axis, which can pass through a fixed position in the axes coordinate, or a fixed position in the data coordinate (i.e., the axis floats around when viewlimit changes).

The axes class, by default, have its xaxis and yaxis invisible, and has 4 additional artists which are responsible to draw axis in “left”, “right”, “bottom” and “top”. They are accessed as ax.axis[“left”], ax.axis[“right”], and so on, i.e., ax.axis is a dictionary that contains artists (note that ax.axis is still a callable methods and it behaves as an original Axes.axis method in mpl).

To create an axes,

```python
import mpl_toolkits.axisartist as AA
fig = plt.figure(1)
ax = AA.Axes(fig, [0.1, 0.1, 0.8, 0.8])
fig.add_axes(ax)
```

or to create a subplot

```python
ax = AA.Subplot(fig, 111)
fig.add_subplot(ax)
```
For example, you can hide the right, and top axis by

```python
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)
```

It is also possible to add an extra axis. For example, you may have an horizontal axis at y=0 (in data coordinate).

```python
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```

```python
import matplotlib.pyplot as plt
import mpl_toolkits.axisartist as AA

fig = plt.figure(1)
fig.subplots_adjust(right=0.85)
ax = AA.Subplot(fig, 1, 1, 1)
fig.add_subplot(ax)

# make some axis invisible
ax.axis["bottom", "top", "right"].set_visible(False)

# make an new axis along the first axis axis (x-axis) which pass througth y=0.
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0,
                                        axis_direction="bottom")

ax.axis["y=0"].toggle(all=True)
ax.axis["y=0"].label.set_text("y = 0")

ax.set_ylim(-2, 4)

plt.show()
```
Or a fixed axis with some offset

```python
# make new (right-side) yaxis, but with some offset
ax.axis['right2'] = ax.new_fixed_axis(loc="right",
                                     offset=(20, 0))
```

**AxesArtist with ParasiteAxes**  Most commands in the axes_grid1 toolkit can take a axes_class keyword argument, and the commands creates an axes of the given class. For example, to create a host subplot with axes_artist.Axes,

```python
import mpl_toolkits.axisartist as AA
from mpl_toolkits.axes_grid1 import host_subplot

host = host_subplot(111, axes_class=AA.Axes)
```

Here is an example that uses parasiteAxes.
Curvilinear Grid  The motivation behind the AxisArtist module is to support curvilinear grid and ticks.
See \texttt{AXISARTIST namespace} for more details.

\textbf{Floating Axes} \: This also support a Floating Axes whose outer axis are defined as floating axis.
AxesDivider

The axes_divider module provides helper classes to adjust the axes positions of a set of images in the drawing time.

- **axes_size** provides a class of units that the size of each axes will be determined. For example, you can specify a fixed size

- **Divider** this is the class that is used to calculate the axes position. It divides the given rectangular area into several areas. You initialize the divider by setting the horizontal and vertical list of sizes that the division will be based on. You then use the `new_locator` method, whose return value is a callable object that can be used to set the `axes_locator` of the axes.

You first initialize the divider by specifying its grids, i.e., horizontal and vertical.

```python
rect = [0.2, 0.2, 0.6, 0.6]
horiz=[h0, h1, h2, h3]
vert=[v0, v1, v2]
divider = Divider(fig, rect, horiz, vert)
```

where, `rect` is a bounds of the box that will be divided and `h0...h3, v0...v2` need to be an instance of classes in the `axes_size`. They have `get_size` method that returns a tuple of two floats. The first float is the relative size, and the second float is the absolute size. Consider a following grid.
The height of the bottom row is always 2 (axes_divider internally assumes that the unit is inch). The first and the second rows with height ratio of 2:3. For example, if the total height of the grid is 6, then the first and second row will each occupy 2/(2+3) and 3/(2+3) of (6-1) inches. The widths of columns (horiz) will be similarly determined. When aspect ratio is set, the total height (or width) will be adjusted accordingly.

The `mpl_toolkits.axes_grid.axes_size` contains several classes that can be used to set the horizontal and vertical configurations. For example, for the vertical configuration above will be:

```python
from mpl_toolkits.axes_grid.axes_size import Fixed, Scaled
vert = [Fixed(2), Scaled(2), Scaled(3)]
```

After you set up the divider object, then you create a locator instance which will be given to the axes:

```python
locator = divider.new_locator(nx=0, ny=1)
ax.set_axes_locator(locator)
```

The return value of the `new_locator` method is a instance of the AxesLocator class. It is a callable object that returns the location and size of the cell at the first column and the second row. You may create a locator that spans over multiple cells:

```python
locator = divider.new_locator(nx=0, nx=2, ny=1)
```

The above locator, when called, will return the position and size of the cells spanning the first and second column and the first row. You may consider it as [0:2, 1].

See the example,

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4.))
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d"%i) for i in range(4)]

horiz = [Size.Scaled(1.5), Size.Fixed(.5), Size.Scaled(1.), Size.Scaled(.5)]
vert = [Size.Scaled(1.), Size.Fixed(.5), Size.Scaled(1.5)]

divider = Divider(fig1, rect, horiz, vert, aspect=False)
```

<table>
<thead>
<tr>
<th>v0</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>h0,v2</td>
<td>h1</td>
</tr>
</tbody>
</table>
You can adjust the size of the each axes according to their x or y data limits (AxesX and AxesY), similar to the axes aspect parameter.

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4))

# the rect parameter will be ignore as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d"%i) for i in range(4)]

horiz = [Size.AxesX(ax[0]), Size.Fixed(.5), Size.AxesX(ax[1])]
```
vert = [Size.AxesY(ax[0]), Size.Fixed(.5), Size.AxesY(ax[2])]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)

ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=2, ny=0))
ax[2].set_axes_locator(divider.new_locator(nx=0, ny=2))
ax[3].set_axes_locator(divider.new_locator(nx=2, ny=2))

ax[0].set_xlim(0, 2)
ax[1].set_xlim(0, 1)
ax[0].set_ylim(0, 1)
ax[2].set_ylim(0, 2)

divider.set_aspect(1.)

for ax1 in ax:
    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(), visible=False)

plt.draw()
plt.show()
AXISARTIST namespace

The AxisArtist namespace includes a derived Axes implementation. The biggest difference is that the artists responsible to draw axis line, ticks, ticklabel and axis labels are separated out from the mpl’s Axis class, which are much more than artists in the original mpl. This change was strongly motivated to support curvilinear grid. Here are a few things that mpl_toolkits.axisartist.Axes is different from original Axes from mpl.

- Axis elements (axis line(spine), ticks, ticklabel and axis labels) are drawn by a AxisArtist instance. Unlike Axis, left, right, top and bottom axis are drawn by separate artists. And each of them may have different tick location and different tick labels.

- gridlines are drawn by a Gridlines instance. The change was motivated that in curvilinear coordinate, a gridline may not cross axis-lines (i.e., no associated ticks). In the original Axes class, gridlines are tied to ticks.

- ticklines can be rotated if necessary (i.e, along the gridlines)

In summary, all these changes was to support

- a curvilinear grid.
- a floating axis
mpl_toolkits.axisartist.Axes class defines a `axis` attribute, which is a dictionary of AxisArtist instances. By default, the dictionary has 4 AxisArtist instances, responsible for drawing of left, right, bottom and top axis. xaxis and yaxis attributes are still available, however they are set to not visible. As separate artists are used for rendering axis, some axis-related method in mpl may have no effect. In addition to AxisArtist instances, the mpl_toolkits.axisartist.Axes will have `gridlines` attribute (Gridlines), which obviously draws grid lines.

In both AxisArtist and Gridlines, the calculation of tick and grid location is delegated to an instance of GridHelper class. mpl_toolkits.axisartist.Axes class uses GridHelperRectlinear as a grid helper. The GridHelperRectlinear class is a wrapper around the `xaxis` and `yaxis` of mpl’s original Axes, and it was meant to work as the way how mpl’s original axes works. For example, tick location changes using set_ticks method and etc. should work as expected. But change in artist properties (e.g., color) will not work in general, although some effort has been made so that some often-change attributes (color, etc.) are respected.

**AxisArtist**  
AxisArtist can be considered as a container artist with following attributes which will draw ticks, labels, etc.

- line
- major_ticks, major_ticklabels
- minor_ticks, minor_ticklabels
- offsetText
- label

**line**  
Derived from Line2d class. Responsible for drawing a spinal(?) line.

**major_ticks, minor_ticks**  
Derived from Line2d class. Note that ticks are markers.

**major_ticklabels, minor_ticklabels**  
Derived from Text. Note that it is not a list of Text artist, but a single artist (similar to a collection).

**axislabel**  
Derived from Text.

**Default AxisArtists**  
By default, following for axis artists are defined:

```python
ax.axis["left"], ax.axis["bottom"], ax.axis["right"], ax.axis["top"]
```

The ticklabels and axislabel of the top and the right axis are set to not visible.

For example, if you want to change the color attributes of major_ticklabels of the bottom x-axis

```python
ax.axis["bottom"].major_ticklabels.set_color("b")
```

Similarly, to make ticklabels invisible

31.2. AxesGrid
ax.axis["bottom"].major_ticklabels.set_visible(False)

AxisArtist provides a helper method to control the visibility of ticks, ticklabels, and label. To make ticklabel invisible,

ax.axis["bottom"].toggle(ticklabels=False)

To make all of ticks, ticklabels, and (axis) label invisible

ax.axis["bottom"].toggle(all=False)

To turn all off but ticks on

ax.axis["bottom"].toggle(all=False, ticks=True)

To turn all on but (axis) label off

ax.axis["bottom"].toggle(all=True, label=False)

ax.axis's __getitem__ method can take multiple axis names. For example, to turn ticklabels of “top” and “right” axis on,

ax.axis["top","right"].toggle(ticklabels=True)

Note that ‘ax.axis["top","right"]’ returns a simple proxy object that translate above code to something like below.

for n in ["top","right"]:
    ax.axis[n].toggle(ticklabels=True)

So, any return values in the for loop are ignored. And you should not use it anything more than a simple method.

Like the list indexing ":" means all items, i.e.,

ax.axis[:].major_ticks.set_color("r")

changes tick color in all axis.

HowTo

1. Changing tick locations and label.
   
   Same as the original mpl’s axes:

   
   ax.set_xticks([1,2,3])

2. Changing axis properties like color, etc.

   Change the properties of appropriate artists. For example, to change the color of the ticklabels:

   
   ax.axis["left"].major_ticklabels.set_color("r")
3. To change the attributes of multiple axis:

   ```python
   ax.axis["left","bottom"].major_ticklabels.set_color("r")
   ```

   or to change the attributes of all axis:

   ```python
   ax.axis[:].major_ticklabels.set_color("r")
   ```

4. To change the tick size (length), you need to use axis.major_ticks.set_ticksize method. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use axis.major_ticks.set_tick_out method.

   To change the pad between ticks and ticklabels, use axis.major_ticklabels.set_pad method.

   To change the pad between ticklabels and axis label, axis.label.set_pad method.

**Rotation and Alignment of TickLabels**  
This is also quite different from the original mpl and can be confusing. When you want to rotate the ticklabels, first consider using "set_axis_direction" method.

```python
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["right"].label.set_axis_direction("left")
```

![Diagram showing rotated ticklabels](image)

The parameter for set_axis_direction is one of ["left", "right", "bottom", "top"].

You must understand some underlying concept of directions.

1. There is a reference direction which is defined as the direction of the axis line with increasing coordinate. For example, the reference direction of the left x-axis is from bottom to top.
The direction, text angle, and alignments of the ticks, ticklabels and axis-label is determined with respect to the reference direction.

2. *ticklabel_direction* is either the right-hand side (+) of the reference direction or the left-hand side (-).

3. same for the *label_direction*
4. Ticks are by default drawn toward the opposite direction of the ticklabels.

5. Text rotation of ticklabels and label is determined in reference to the `ticklabel_direction` or `label_direction`, respectively. The rotation of ticklabels and label is anchored.

On the other hand, there is a concept of “axis_direction”. This is a default setting of above properties for each, “bottom”, “left”, “top”, and “right” axis.

<table>
<thead>
<tr>
<th>axis label</th>
<th>direction</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axis label</td>
<td>rotation</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axis label</td>
<td>va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axis label</td>
<td>ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>tick label</td>
<td>direction</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>tick labels</td>
<td>rotation</td>
<td>0</td>
<td>center</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>tick label</td>
<td>ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>tick label</td>
<td>va</td>
<td>center</td>
<td>baseline</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And, `set_axis_direction(“top”)` means to adjust the text rotation etc, for settings suitable for “top” axis. The concept of axis direction can be more clear with curved axis.
The axis_direction can be adjusted in the AxisArtist level, or in the level of its child artists, i.e., ticks, ticklabels, and axis-label.

```python
ax1.axis['left'].set_axis_direction("top")
```
changes axis_direction of all the associated artist with the “left” axis, while

```python
ax1.axis['left'].major_ticklabels.set_axis_direction("top")
```
changes the axis_direction of only the major_ticklabels. Note that set_axis_direction in the AxisArtist level changes the ticklabel_direction and label_direction, while changing the axis_direction of ticks, ticklabels, and axis-label does not affect them.

If you want to make ticks outward and ticklabels inside the axes, use invert_ticklabel_direction method.

```python
ax.axis[:].invert_ticklabel_direction()
```

A related method is “set_tick_out”. It makes ticks outward (as a matter of fact, it makes ticks toward the opposite direction of the default direction).

```python
ax.axis[:].major_ticks.set_tick_out(True)
```
So, in summary,

- **AxisArtist’s methods**
  - `set_axis_direction`: “left”, “right”, “bottom”, or “top”
  - `set_ticklabel_direction`: “+” or “-”
  - `set_axislabel_direction`: “+” or “-”
  - `invert_ticklabel_direction`

- **Ticks’ methods (major_ticks and minor_ticks)**
  - `set_tick_out`: True or False
  - `set_ticksize`: size in points

- **TickLabels’ methods (major_ticklabels and minor_ticklabels)**
  - `set_axis_direction`: “left”, “right”, “bottom”, or “top”
  - `set_rotation`: angle with respect to the reference direction
  - `set_ha` and `set_va`: see below

- **AxisLabels’ methods (label)**
  - `set_axis_direction`: “left”, “right”, “bottom”, or “top”
  - `set_rotation`: angle with respect to the reference direction
  - `set_ha` and `set_va`

**Adjusting ticklabels alignment** Alignment of TickLabels are treated specially. See below
Adjusting pad  To change the pad between ticks and ticklabels

ax.axis["left"].major_ticklabels.set_pad(10)

Or ticklabels and axis-label

ax.axis["left"].label.set_pad(10)
GridHelper  To actually define a curvilinear coordinate, you have to use your own grid helper. A generalised version of grid helper class is supplied and this class should suffice in most of cases. A user may provide two functions which defines a transformation (and its inverse pair) from the curved coordinate to (rectilinear) image coordinate. Note that while ticks and grids are drawn for curved coordinate, the data transform of the axes itself (ax.transData) is still rectilinear (image) coordinate.

```python
from mpl_toolkits.axisartist.grid_helper_curvelinear import GridHelperCurveLinear
from mpl_toolkits.axisartist import Subplot

# from curved coordinate to rectlinear coordinate.
def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

# from rectlinear coordinate to curved coordinate.
def inv_tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y+x

grid_helper = GridHelperCurveLinear((tr, inv_tr))

ax1 = Subplot(fig, 1, 1, 1, grid_helper=grid_helper)

fig.add_subplot(ax1)
```

You may use matplotlib’s Transform instance instead (but a inverse transformation must be defined). Often, coordinate range in a curved coordinate system may have a limited range, or may have cycles. In those cases, a more customized version of grid helper is required.

```python
import mpl_toolkits.axisartist.angle_helper as angle_helper

# PolarAxes.PolarTransform takes radian. However, we want our coordinate
# system in degree
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

# extreme finder : find a range of coordinate.
# 20, 20 : number of sampling points along x, y direction
# The first coordinate (longitude, but theta in polar)
# has a cycle of 360 degree.
# The second coordinate (latitude, but radius in polar) has a minimum of 0
extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
    lon_cycle=360,
    lat_cycle=None,
    lon_minmax=None,
    lat_minmax=(0, np.inf),
)

# Find a grid values appropriate for the coordinate (degree,
# minute, second). The argument is a approximate number of grids.
grid_locator1 = angle_helper.LocatorDMS(12)
```

31.2. AxesGrid
# And also uses an appropriate formatter. Note that, the
# acceptable Locator and Formatter class is a bit different than
# that of mpl's, and you cannot directly use mpl's Locator and
# Formatter here (but may be possible in the future).
tick_formatter1 = angle_helper.FormatterDMS()

grid_helper = GridHelperCurveLinear(tr,
    extreme_finder=extreme_finder,
    grid_locator1=grid_locator1,
    tick_formatter1=tick_formatter1
)

Again, the transData of the axes is still a rectilinear coordinate (image coordinate). You may manually do
conversion between two coordinates, or you may use Parasite Axes for convenience.:

ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

# A parasite axes with given transform
ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anthing you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)

FloatingAxis A floating axis is an axis one of whose data coordinate is fixed, i.e., its location is not
fixed in Axes coordinate but changes as axes data limits changes. A floating axis can be created using
new_floating_axis method. However, it is your responsibility that the resulting AxisArtist is properly added
to the axes. A recommended way is to add it as an item of Axes’s axis attribute.:
# floating axis whose first (index starts from 0) coordinate
# (theta) is fixed at 60

ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^\circ$")
axis.label.set_visible(True)

See the first example of this page.

**Current Limitations and TODO’s**  
The code need more refinement. Here is an incomplete list of issues and TODO’s

- No easy way to support a user customized tick location (for curvilinear grid). A new Locator class needs to be created.
- FloatingAxis may have coordinate limits, e.g., a floating axis of x = 0, but y only spans from 0 to 1.
- The location of axislabel of FloatingAxis needs to be optionally given as a coordinate value. ex, a floating axis of x=0 with label at y=1

**The Matplotlib AxesGrid Toolkit API**

- **Release** 1.4.3  
  **Date** February 16, 2015

```python
mpl_toolkits.axes_grid.axes_size

class mpl_toolkits.axes_grid.axes_size.Fixed(fixed_size)
    Simple fixed size with absolute part = fixed_size and relative part = 0

class mpl_toolkits.axes_grid.axes_size.Scaled(scalable_size)
    Simple scaled(size) with absolute part = 0 and relative part = scalable_size

class mpl_toolkits.axes_grid.axes_size.AxesX(axes, aspect=1.0, ref_ax=None)
    Scaled size whose relative part corresponds to the data width of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.AxesY(axes, aspect=1.0, ref_ax=None)
    Scaled size whose relative part corresponds to the data height of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.MaxWidth(artist_list)
    Size whose absolute part is the largest width of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.MaxHeight(artist_list)
    Size whose absolute part is the largest height of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.Fraction(fraction, ref_size)
    An instance whose size is a fraction of the ref_size.

>>> s = Fraction(0.3, AxesX(ax))
```
class `mpl_toolkits.axes_grid.axes_size.Padded`(*size*, *pad*)

Return a instance where the absolute part of *size* is increase by the amount of *pad*.

`mpl_toolkits.axes_grid.axes_size.from_any`(*size*, *fraction_ref=None*)

Creates Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %. The second argument is only meaningful when Fraction unit is created.:  

```python
g = Size.from_any(1.2) # => Size.Fixed(1.2)
g = Size.from_any("50\%", a) # => Size.Fraction(0.5, a)
```

**mpl_toolkits.axes_grid.axes_divider**

class `mpl_toolkits.axes_grid.axes_divider.Divider`(*fig*, *pos*, *horizontal*, *vertical*, *aspect=None*, *anchor=u'C'*)

This is the class that is used calculates the axes position. It divides the given rectangular area into several sub-rectangles. You initialize the divider by setting the horizontal and vertical lists of sizes (`mpl_toolkits.axes_grid.axes_size`) that the division will be based on. You then use the new_locator method to create a callable object that can be used as the axes_locator of the axes.

**Parameters**

- *fig* – matplotlib figure
- *pos* – position (tuple of 4 floats) of the rectangle that will be divided.
- *horizontal* – list of sizes (`axes_size`) for horizontal division
- *vertical* – list of sizes (`axes_size`) for vertical division
- *aspect* – if True, the overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.
- *anchor* – Determine how the reduced rectangle is placed when aspect is True.

`add_auto_adjustable_area`(*use_axes*, *pad=0.1*, *adjust_dirs=None*)

`append_size`(*position*, *size*)

`get_anchor`()

return the anchor

`get_aspect`()

return aspect

`get_horizontal`()

return horizontal sizes

`get_horizontal_sizes`(*renderer*)

`get_locator`()

`get_position`()

return the position of the rectangle.
get_position_runtime(ax, renderer)

get_vertical()
    return vertical sizes

get_vertical_sizes(renderer)

get_vsize_hsize()

locate(nx, ny, nx1=None, ny1=None, axes=None, renderer=None)

Parameters
- nx1 (nx,) – Integers specifying the column-position of the cell. When nx1 is None, a single nx-th column is specified. Otherwise location of columns spanning between nx to nx1 (but excluding nx1-th column) is specified.
- ny1 (ny,) – same as nx and nx1, but for row positions.

new_locator(nx, ny, nx1=None, ny1=None)
    returns a new locator (mpl_toolkits.axes_grid.axes_divider.AxesLocator) for specified cell.

Parameters
- nx1 (nx,) – Integers specifying the column-position of the cell. When nx1 is None, a single nx-th column is specified. Otherwise location of columns spanning between nx to nx1 (but excluding nx1-th column) is specified.
- ny1 (ny,) – same as nx and nx1, but for row positions.

set_anchor(ANCHOR)

Parameters anchor
- anchor position

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>’C’</td>
<td>Center</td>
</tr>
<tr>
<td>’SW’</td>
<td>bottom left</td>
</tr>
<tr>
<td>’S’</td>
<td>bottom</td>
</tr>
<tr>
<td>’SE’</td>
<td>bottom right</td>
</tr>
<tr>
<td>’E’</td>
<td>right</td>
</tr>
<tr>
<td>’NE’</td>
<td>top right</td>
</tr>
<tr>
<td>’N’</td>
<td>top</td>
</tr>
<tr>
<td>’NW’</td>
<td>top left</td>
</tr>
<tr>
<td>’W’</td>
<td>left</td>
</tr>
</tbody>
</table>

set_aspect(aspect=False)

Parameters anchor
- True or False

set_horizontal(h)

Parameters horizontal
- list of sizes (axes_size) for horizontal division
**set locator**(_locator_)

**set position**(pos)
set the position of the rectangle.

Parameters pos – position (tuple of 4 floats) of the rectangle that will be divided.

**set_vertical**(v)

Parameters horizontal – list of sizes (axes_size) for horizontal division

**class** mpl_toolkits.axes_grid.axes_divider.AxesLocator(axes_divider, nx, ny, nx1=None, ny1=None)
A simple callable object, initialized with AxesDivider class, returns the position and size of the given cell.

Parameters

• axes_divider – An instance of AxesDivider class.
• nx1 (nx,) – Integers specifying the column-position of the cell. When nx1 is None, a single nx-th column is specified. Otherwise location of columns spanning between nx to nx1 (but excluding nx1-th column) is is specified.
• ny1 (ny,) – same as nx and nx1, but for row positions.

**get subplotspec**()

**class** mpl_toolkits.axes_grid.axes_divider.SubplotDivider(fig, *args, **kwargs)
The Divider class whose rectangle area is specified as a subplot geometry.

fig is a matplotlib.figure.Figure instance.

args is the tuple (numRows, numCols, plotNum), where the array of subplots in the figure has dimensions numRows, numCols, and where plotNum is the number of the subplot being created. plotNum starts at 1 in the upper left corner and increases to the right.

If numRows <= numCols <= plotNum < 10, args can be the decimal integer numRows * 100 + numCols * 10 + plotNum.

**change geometry**(numrows, numcols, num)
change subplot geometry, e.g., from 1,1,1 to 2,2,3

**get geometry**()
get the subplot geometry, e.g., 2,2,3

**get position**()
return the bounds of the subplot box

**get subplotspec**()
get the SubplotSpec instance

**set subplotspec**(subplotspec)
set the SubplotSpec instance

**update params**()
update the subplot position from fig.subplotpars
class mpl_toolkits.axes_grid.axes_divider.AxesDivider(axes, xref=None, yref=None)
Divider based on the pre-existing axes.

Parameters

axes -- axes

append_axes(position, size, pad=None, add_to_figure=True, **kwargs)
create an axes at the given position with the same height (or width) of the main axes.

position ['"left"'|"right"|"bottom"|"top"]

size and pad should be axes_grid.axes_size compatible.

new_horizontal(size, pad=None, pack_start=False, **kwargs)
Add a new axes on the right (or left) side of the main axes.

Parameters

• size -- A width of the axes. A axes_size instance or if float or string is given, from_any
function is used to create one, with ref_size set to AxesX instance of the current axes.
• pad -- pad between the axes. It takes same argument as size.
• pack_start -- If False, the new axes is appended at the end of the list, i.e., it became the
right-most axes. If True, it is inserted at the start of the list, and becomes the left-most
axes.

All extra keywords arguments are passed to the created axes. If axes_class is given, the new axes
will be created as an instance of the given class. Otherwise, the same class of the main axes will be
used.

new_vertical(size, pad=None, pack_start=False, **kwargs)
Add a new axes on the top (or bottom) side of the main axes.

Parameters

• size -- A height of the axes. A axes_size instance or if float or string is given, from_any
function is used to create one, with ref_size set to AxesX instance of the current axes.
• pad -- pad between the axes. It takes same argument as size.
• pack_start -- If False, the new axes is appended at the end of the list, i.e., it became the
top-most axes. If True, it is inserted at the start of the list, and becomes the bottom-most
axes.

All extra keywords arguments are passed to the created axes. If axes_class is given, the new axes
will be created as an instance of the given class. Otherwise, the same class of the main axes will be
used.

mpl_toolkits.axes_grid.axes_grid

class mpl_toolkits.axes_grid.axes_grid.Grid(fig, rect, nrows_ncols, ngrids=None,
direction=u'row', axes_pad=0.02, add_all=True, share_all=False, share_x=True,
share_y=True, label_mode=u'L', axes_class=None)

Build an Grid instance with a grid nrows*ncols Axes in Figure fig with rect=[left, bottom, width, height] (in Figure coordinates) or the subplot position code (e.g., “121”).

Optional keyword arguments:
Matplotlib, Release 1.4.3

<table>
<thead>
<tr>
<th>Key-</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>direction</td>
<td>“row”</td>
<td>[ “row”</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>share_x</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_y</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[ “L”</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td>a type object which must be a subclass of Axes</td>
</tr>
</tbody>
</table>

```python
class mpl_toolkits.axes_grid.axes_grid.ImageGrid(fig, rect, nrows_ncols, ngids=None, direction=u’row’, axes_pad=0.02, add_all=True, share_all=False, aspect=True, label_mode=u’L’, cbar_mode=None, cbar_location=u’right’, cbar_pad=None, cbar_size=u’5%’, cbar_set_cax=True, axes_class=None)
```

Build an ImageGrid instance with a grid nrows*ncols Axes in Figure fig with rect=[left, bottom, width, height] (in Figure coordinates) or the subplot position code (e.g., “121”).

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>“row”</td>
<td>[ “row”</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[ “L”</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>[ “each”</td>
</tr>
<tr>
<td>cbar_location</td>
<td>“right”</td>
<td>[ “left”</td>
</tr>
<tr>
<td>cbar_pad</td>
<td>None</td>
<td>[ True</td>
</tr>
<tr>
<td>cbar_size</td>
<td>“5%”</td>
<td>[ True</td>
</tr>
<tr>
<td>cbar_set_cax</td>
<td>True</td>
<td>a type object which must be a subclass of axes_grid’s subclass of Axes</td>
</tr>
</tbody>
</table>

**cbar_set_cax** [if True, each axes in the grid has a cax] attribute that is bind to associated cbar_axes.
mpl_toolkits.axes_grid.axis_artist

class mpl_toolkits.axes_grid.axis_artist.AxisArtist(axes, helper, offset=None, axis_direction=u'bottom', **kw)

An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, ticklabels, and axis label.

axes : axes helper : an AxisArtistHelper instance.

LABELPAD

ZORDER = 2.5

draw(artist, renderer, *args, **kwarg)

Draw the axis lines, tick lines and labels

get_axisline_style()

return the current axisline style.

get_helper()

Return axis artist helper instance.

get_tightbbox(renderer)

get_transform()

invert_ticklabel_direction()

set_axis_direction(axis_direction)

Adjust the direction, text angle, text alignment of ticklabels, labels following the matplotlib convention for the rectangle axes.

The axis_direction must be one of [left, right, bottom, top].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels location</td>
<td>“-”</td>
<td>“+”</td>
<td>“+”</td>
<td>“-”</td>
</tr>
<tr>
<td>axislabel location</td>
<td>“-”</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>0</td>
<td>180</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>center</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the direction “+” and “-” are relative to the direction of the increasing coordinate. Also, the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_axislabel_direction(label_direction)

Adjust the direction of the axislabel.
ACCEPTS: [“+” | “-” ]  
Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

set_axisline_style(axisline_style=None, **kw)
Set the axisline style. 
axisline_style can be a string with axisline style name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_arrowstyle("->", size=1.5) set_arrowstyle("->", size=1.5)
Old attrs simply are forgotten.

Without argument (or with arrowstyle=none), return available styles as a list of strings.

set_label(s)

set_ticklabel_direction(tick_direction)
Adjust the direction of the ticklabel. 
ACCEPTS: [“+” | “-” ]  
Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

toggle(all=None, ticks=None, ticklabels=None, label=None)
Toggle visibility of ticks, ticklabels, and (axis) label. To turn all off,

axis.toggle(all=False)
To turn all off but ticks on

axis.toggle(all=False, ticks=True)
To turn all on but (axis) label off

axis.toggle(all=True, label=False))

class mpl_toolkits.axes_grid.axis_artist.Ticks(ticksize, tick_out=False, **kwargs)
Ticks are derived from Line2D, and note that ticks themselves are markers. Thus, you should use set_mec, set_mew, etc.

To change the tick size (length), you need to use set_ticksize. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use set_tick_out(False).

get_tick_out()
Return True if the tick will be rotated by 180 degree.

get_ticksize()
Return length of the ticks in points.

set_tick_out(b)
set True if tick need to be rotated by 180 degree.

set_ticksize(ticksize)
set length of the ticks in points.
class mpl_toolkits.axes_grid.axis_artist.AxisLabel(*kl, **kwargs)

Axis Label. Derived from Text. The position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text.

To change the pad between ticklabels and axis label, use set_pad.

get_pad()

return pad in points. See set_pad for more details.

set_axis_direction(d)

Adjust the text angle and text alignment of axis label according to the matplotlib convention.

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>center</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>axislabel va</td>
<td>top</td>
<td>center</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_pad(pad)

Set the pad in points. Note that the actual pad will be the sum of the internal pad and the external pad (that are set automatically by the AxisArtist), and it only set the internal pad

class mpl_toolkits.axes_grid.axis_artist.TickLabels(**kwargs)

Tick Labels. While derived from Text, this single artist draws all ticklabels. As in AxisLabel, the position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text. Unlike the ticklabels of the mainline matplotlib, properties of single ticklabel alone cannot modified.

To change the pad between ticks and ticklabels, use set_pad.

get_texts_widths_heights_descents(renderer)

return a list of width, height, descent for ticklabels.

set_axis_direction(label_direction)

Adjust the text angle and text alignment of ticklabels according to the matplotlib convention.

The label_direction must be one of [left, right, bottom, top].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>center</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

The matplotlib AxesGrid toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. The AxesGrid toolkit is distributed with matplotlib source.
31.3 MplDataCursor

*(Not distributed with matplotlib)*

MplDataCursor is a toolkit written by Joe Kington to provide interactive “data cursors” (clickable annotation boxes) for matplotlib.

31.4 GTK Tools

mpl_toolkits.gtktools provides some utilities for working with GTK. This toolkit ships with matplotlib, but requires pygtk.

31.5 Excel Tools

mpl_toolkits.exceltools provides some utilities for working with Excel. This toolkit ships with matplotlib, but requires xlwt.

31.6 Natgrid

*(Not distributed with matplotlib)*

mpl_toolkits.natgrid is an interface to natgrid C library for gridding irregularly spaced data. This requires a separate installation of the natgrid toolkit from the sourceforge download page.
Several projects have started to provide a higher-level interface to matplotlib. These are independent projects.

32.1 seaborn

*(Not distributed with matplotlib)*

seaborn is a high level interface for drawing statistical graphics with matplotlib. It aims to make visualization a central part of exploring and understanding complex datasets.

32.2 ggplot

*(Not distributed with matplotlib)*

ggplot is a port of the R ggplot2 to python based on matplotlib.

32.3 prettyplotlib

*(Not distributed with matplotlib)*

prettyplotlib is an extension to matplotlib which changes many of the defaults to make plots some consider more attractive.
Part VIII

The Matplotlib API
## PLOTTING COMMANDS SUMMARY

```python
matplotlib.pyplot.plotting()
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acorr</td>
<td>Plot the autocorrelation of $x$.</td>
</tr>
<tr>
<td>angle_spectrum</td>
<td>Plot the angle spectrum.</td>
</tr>
<tr>
<td>annotate</td>
<td>Create an annotation: a piece of text referring to a data point.</td>
</tr>
<tr>
<td>arrow</td>
<td>Add an arrow to the axes.</td>
</tr>
<tr>
<td>autoscale</td>
<td>Autoscale the axis view to the data (toggle).</td>
</tr>
<tr>
<td>axes</td>
<td>Add an axes to the figure.</td>
</tr>
<tr>
<td>axhline</td>
<td>Add a horizontal line across the axis.</td>
</tr>
<tr>
<td>axhspan</td>
<td>Add a horizontal span (rectangle) across the axis.</td>
</tr>
<tr>
<td>axis</td>
<td>Convenience method to get or set axis properties.</td>
</tr>
<tr>
<td>axvline</td>
<td>Add a vertical line across the axes.</td>
</tr>
<tr>
<td>axvspan</td>
<td>Add a vertical span (rectangle) across the axes.</td>
</tr>
<tr>
<td>bar</td>
<td>Make a bar plot.</td>
</tr>
<tr>
<td>barbs</td>
<td>Plot a 2-D field of barbs.</td>
</tr>
<tr>
<td>barh</td>
<td>Make a horizontal bar plot.</td>
</tr>
<tr>
<td>box</td>
<td>Turn the axes box on or off.</td>
</tr>
<tr>
<td>boxplot</td>
<td>Make a box and whisker plot.</td>
</tr>
<tr>
<td>broken_barh</td>
<td>Plot horizontal bars.</td>
</tr>
<tr>
<td>cla</td>
<td>Clear the current axes.</td>
</tr>
<tr>
<td>clabel</td>
<td>Label a contour plot.</td>
</tr>
<tr>
<td>clf</td>
<td>Clear the current figure.</td>
</tr>
<tr>
<td>clim</td>
<td>Set the color limits of the current image.</td>
</tr>
<tr>
<td>close</td>
<td>Close a figure window.</td>
</tr>
<tr>
<td>cohere</td>
<td>Plot the coherence between $x$ and $y$.</td>
</tr>
<tr>
<td>colorbar</td>
<td>Add a colorbar to a plot.</td>
</tr>
<tr>
<td>contour</td>
<td>Plot contours.</td>
</tr>
<tr>
<td>contourf</td>
<td>Plot contours.</td>
</tr>
<tr>
<td>csd</td>
<td>Plot the cross-spectral density.</td>
</tr>
<tr>
<td>delaxes</td>
<td>Remove an axes from the current figure.</td>
</tr>
<tr>
<td>draw</td>
<td>Redraw the current figure.</td>
</tr>
<tr>
<td>errorbar</td>
<td>Plot an errorbar graph.</td>
</tr>
<tr>
<td>eventplot</td>
<td>Plot identical parallel lines at specific positions.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figimage</td>
<td>Adds a non-resampled image to the figure.</td>
</tr>
<tr>
<td>figlegend</td>
<td>Place a legend in the figure.</td>
</tr>
<tr>
<td>figtext</td>
<td>Add text to figure.</td>
</tr>
<tr>
<td>figure</td>
<td>Creates a new figure.</td>
</tr>
<tr>
<td>fill</td>
<td>Plot filled polygons.</td>
</tr>
<tr>
<td>fill_between</td>
<td>Make filled polygons between two curves.</td>
</tr>
<tr>
<td>fill_betweenx</td>
<td>Make filled polygons between two horizontal curves.</td>
</tr>
<tr>
<td>findobj</td>
<td>Find artist objects.</td>
</tr>
<tr>
<td>gca</td>
<td>Get the current Axes instance on the current figure matching the given keyword args, or create one.</td>
</tr>
<tr>
<td>gcf</td>
<td>Get a reference to the current figure.</td>
</tr>
<tr>
<td>gci</td>
<td>Get the current colorable artist.</td>
</tr>
<tr>
<td>get_figlabels</td>
<td>Return a list of existing figure labels.</td>
</tr>
<tr>
<td>get_fignums</td>
<td>Return a list of existing figure numbers.</td>
</tr>
<tr>
<td>grid</td>
<td>Turn the axes grids on or off.</td>
</tr>
<tr>
<td>hexbin</td>
<td>Make a hexagonal binning plot.</td>
</tr>
<tr>
<td>hist</td>
<td>Plot a histogram.</td>
</tr>
<tr>
<td>hist2d</td>
<td>Make a 2D histogram plot.</td>
</tr>
<tr>
<td>hlines</td>
<td>Plot horizontal lines at each y from xmin to xmax.</td>
</tr>
<tr>
<td>hold</td>
<td>Set the hold state.</td>
</tr>
<tr>
<td>imread</td>
<td>Read an image from a file into an array.</td>
</tr>
<tr>
<td>imsave</td>
<td>Save an array as in image file.</td>
</tr>
<tr>
<td>imshow</td>
<td>Display an image on the axes.</td>
</tr>
<tr>
<td>ioff</td>
<td>Turn interactive mode off.</td>
</tr>
<tr>
<td>ion</td>
<td>Turn interactive mode on.</td>
</tr>
<tr>
<td>ishold</td>
<td>Return the hold status of the current axes.</td>
</tr>
<tr>
<td>isinteractive</td>
<td>Return status of interactive mode.</td>
</tr>
<tr>
<td>legend</td>
<td>Places a legend on the axes.</td>
</tr>
<tr>
<td>locator_params</td>
<td>Control behavior of tick locators.</td>
</tr>
<tr>
<td>loglog</td>
<td>Make a plot with log scaling on both the x and y axis.</td>
</tr>
<tr>
<td>magnitude_spectrum</td>
<td>Plot the magnitude spectrum.</td>
</tr>
<tr>
<td>margins</td>
<td>Set or retrieve autoscaling margins.</td>
</tr>
<tr>
<td>matshow</td>
<td>Display an array as a matrix in a new figure window.</td>
</tr>
<tr>
<td>minorticks_off</td>
<td>Remove minor ticks from the current plot.</td>
</tr>
<tr>
<td>minorticks_on</td>
<td>Display minor ticks on the current plot.</td>
</tr>
<tr>
<td>over</td>
<td>Call a function with hold(True).</td>
</tr>
<tr>
<td>pause</td>
<td>Pause for interval seconds.</td>
</tr>
<tr>
<td>pcolor</td>
<td>Create a pseudocolor plot of a 2-D array.</td>
</tr>
<tr>
<td>pcolormesh</td>
<td>Plot a quadrilateral mesh.</td>
</tr>
<tr>
<td>phase_spectrum</td>
<td>Plot the phase spectrum.</td>
</tr>
<tr>
<td>pie</td>
<td>Plot a pie chart.</td>
</tr>
<tr>
<td>plot</td>
<td>Plot lines and/or markers to the Axes.</td>
</tr>
<tr>
<td>plot_date</td>
<td>Plot with data with dates.</td>
</tr>
<tr>
<td>plotfile</td>
<td>Plot the data in in a file.</td>
</tr>
<tr>
<td>polar</td>
<td>Make a polar plot.</td>
</tr>
<tr>
<td>psd</td>
<td>Plot the power spectral density.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>quiver</td>
<td>Plot a 2-D field of arrows.</td>
</tr>
<tr>
<td>quiverkey</td>
<td>Add a key to a quiver plot.</td>
</tr>
<tr>
<td>rc</td>
<td>Set the current rc params.</td>
</tr>
<tr>
<td>rc_context</td>
<td>Return a context manager for managing rc settings.</td>
</tr>
<tr>
<td>rcdefaults</td>
<td>Restore the default rc params.</td>
</tr>
<tr>
<td>rgrids</td>
<td>Get or set the radial gridlines on a polar plot.</td>
</tr>
<tr>
<td>savefig</td>
<td>Save the current figure.</td>
</tr>
<tr>
<td>sca</td>
<td>Set the current Axes instance to ax.</td>
</tr>
<tr>
<td>scatter</td>
<td>Make a scatter plot of x vs y, where x and y are sequence like objects of the same lengths.</td>
</tr>
<tr>
<td>sci</td>
<td>Set the current image.</td>
</tr>
<tr>
<td>semilogx</td>
<td>Make a plot with log scaling on the x axis.</td>
</tr>
<tr>
<td>semilogy</td>
<td>Make a plot with log scaling on the y axis.</td>
</tr>
<tr>
<td>set_cmap</td>
<td>Set the default colormap.</td>
</tr>
<tr>
<td>setp</td>
<td>Set a property on an artist object.</td>
</tr>
<tr>
<td>show</td>
<td>Display a figure.</td>
</tr>
<tr>
<td>specgram</td>
<td>Plot a spectrogram.</td>
</tr>
<tr>
<td>spy</td>
<td>Plot the sparsity pattern on a 2-D array.</td>
</tr>
<tr>
<td>stackplot</td>
<td>Draws a stacked area plot.</td>
</tr>
<tr>
<td>stem</td>
<td>Create a stem plot.</td>
</tr>
<tr>
<td>step</td>
<td>Make a step plot.</td>
</tr>
<tr>
<td>streamplot</td>
<td>Draws streamlines of a vector flow.</td>
</tr>
<tr>
<td>subplot</td>
<td>Return a subplot axes positioned by the given grid definition.</td>
</tr>
<tr>
<td>subplot2grid</td>
<td>Create a subplot in a grid.</td>
</tr>
<tr>
<td>subplot_tool</td>
<td>Launch a subplot tool window for a figure.</td>
</tr>
<tr>
<td>subplots</td>
<td>Create a figure with a set of subplots already made.</td>
</tr>
<tr>
<td>subplots_adjust</td>
<td>Tune the subplot layout.</td>
</tr>
<tr>
<td>suptitle</td>
<td>Add a centered title to the figure.</td>
</tr>
<tr>
<td>switch_backend</td>
<td>Switch the default backend.</td>
</tr>
<tr>
<td>table</td>
<td>Add a table to the current axes.</td>
</tr>
<tr>
<td>text</td>
<td>Add text to the axes.</td>
</tr>
<tr>
<td>thetagrids</td>
<td>Get or set the theta locations of the gridlines in a polar plot.</td>
</tr>
<tr>
<td>tick_params</td>
<td>Change the appearance of ticks and tick labels.</td>
</tr>
<tr>
<td>ticklabel_format</td>
<td>Change the ScalarFormatter used by default for linear axes.</td>
</tr>
<tr>
<td>tight_layout</td>
<td>Automatically adjust subplot parameters to give specified padding.</td>
</tr>
<tr>
<td>title</td>
<td>Set a title of the current axes.</td>
</tr>
<tr>
<td>tricontour</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>tricontourf</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>tricontourf</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>tricontourf</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>tricontourf</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>tricontourf</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>triplot</td>
<td>Draw a unstructured triangular grid as lines and/or markers.</td>
</tr>
<tr>
<td>twinx</td>
<td>Make a second axes that shares the x-axis.</td>
</tr>
<tr>
<td>twiny</td>
<td>Make a second axes that shares the y-axis.</td>
</tr>
<tr>
<td>violinplot</td>
<td>Make a violin plot.</td>
</tr>
<tr>
<td>vlines</td>
<td>Plot vertical lines.</td>
</tr>
<tr>
<td>xcorr</td>
<td>Plot the cross correlation between x and y.</td>
</tr>
<tr>
<td>xkcd</td>
<td>Turns on xkcd sketch-style drawing mode.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>xlabel</td>
<td>Set the x axis label of the current axis.</td>
</tr>
<tr>
<td>xlim</td>
<td>Get or set the x limits of the current axes.</td>
</tr>
<tr>
<td>xscale</td>
<td>Set the scaling of the x-axis.</td>
</tr>
<tr>
<td>xticks</td>
<td>Get or set the x-limits of the current tick locations and labels.</td>
</tr>
<tr>
<td>ylabel</td>
<td>Set the y axis label of the current axis.</td>
</tr>
<tr>
<td>ylim</td>
<td>Get or set the y-limits of the current axes.</td>
</tr>
<tr>
<td>yscale</td>
<td>Set the scaling of the y-axis.</td>
</tr>
<tr>
<td>yticks</td>
<td>Get or set the y-limits of the current tick locations and labels.</td>
</tr>
</tbody>
</table>

**matplotlib.pyplot.colormaps()**

Matplotlib provides a number of colormaps, and others can be added using `register_cmap()`. This function documents the built-in colormaps, and will also return a list of all registered colormaps if called.

You can set the colormap for an image, pcolor, scatter, etc, using a keyword argument:

```python
imshow(X, cmap=cm.hot)
```

or using the `set_cmap()` function:

```python
imshow(X)
pyplot.set_cmap('hot')
pyplot.set_cmap('jet')
```

In interactive mode, `set_cmap()` will update the colormap post-hoc, allowing you to see which one works best for your data.

All built-in colormaps can be reversed by appending `_r`: For instance, `gray_r` is the reverse of `gray`.

There are several common color schemes used in visualization:

**Sequential schemes** for unipolar data that progresses from low to high

**Diverging schemes** for bipolar data that emphasizes positive or negative deviations from a central value

**Cyclic schemes** meant for plotting values that wrap around at the endpoints, such as phase angle, wind direction, or time of day

**Qualitative schemes** for nominal data that has no inherent ordering, where color is used only to distinguish categories

The base colormaps are derived from those of the same name provided with Matlab:
<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autumn bone</td>
<td>sequential increasing black-white color map with a tinge of blue, to emulate X-ray film</td>
</tr>
<tr>
<td>cool copper</td>
<td>linearly-decreasing shades of cyan-magenta</td>
</tr>
<tr>
<td>flag</td>
<td>repetitive red-white-blue-black pattern (not cyclic at endpoints)</td>
</tr>
<tr>
<td>gray</td>
<td>sequential linearly-increasing black-to-white grayscale</td>
</tr>
<tr>
<td>hot</td>
<td>sequential black-red-yellow-white, to emulate blackbody radiation from an object at increasing temperatures</td>
</tr>
<tr>
<td>hsv</td>
<td>cyclic red-yellow-green-cyan-blue-magenta-red, formed by changing the hue component in the HSV color space</td>
</tr>
<tr>
<td>jet</td>
<td>a spectral map with dark endpoints, blue-cyan-yellow-red; based on a fluid-jet simulation by NCSA</td>
</tr>
<tr>
<td>pink</td>
<td>sequential increasing pastel black-pink-white, meant for sepia tone colorization of photographs</td>
</tr>
<tr>
<td>prism spring</td>
<td>repetitive red-yellow-green-blue-purple-....green pattern (not cyclic at endpoints)</td>
</tr>
<tr>
<td>summer</td>
<td>sequential linearly-increasing shades of magenta-yellow</td>
</tr>
<tr>
<td>winter</td>
<td>linearly-increasing shades of blue-green</td>
</tr>
</tbody>
</table>

For the above list only, you can also set the colormap using the corresponding `pylab` shortcut interface function, similar to Matlab:

```python
imshow(X)
hot()
jet()
```

The next set of palettes are from the Yorick scientific visualisation package, an evolution of the GIST package, both by David H. Munro:

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gist_earth</td>
<td>mapmaker’s colors from dark blue deep ocean to green lowlands to brown highlands to white mountains</td>
</tr>
<tr>
<td>gist_heat</td>
<td>sequential increasing black-red-orange-white, to emulate blackbody radiation from an iron bar as it grows hotter</td>
</tr>
<tr>
<td>gist_ncar</td>
<td>pseudo-spectral black-blue-green-yellow-red-purple-white colormap from National Center for Atmospheric Research</td>
</tr>
<tr>
<td>gist_rainbow</td>
<td>runs through the colors in spectral order from red to violet at full saturation (like hsv but not cyclic)</td>
</tr>
<tr>
<td>gist_stern</td>
<td>“Stern special” color table from Interactive Data Language software</td>
</tr>
</tbody>
</table>

1 Rainbow colormaps, jet in particular, are considered a poor choice for scientific visualization by many researchers: Rainbow Color Map (Still) Considered Harmful
2 Resembles “BkBlAqGrYeOrReViWh200” from NCAR Command Language. See Color Table Gallery
The following colormaps are based on the ColorBrewer color specifications and designs developed by Cynthia Brewer:

ColorBrewer Diverging (luminance is highest at the midpoint, and decreases towards differently-colored endpoints):

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrBG</td>
<td>brown, white, blue-green</td>
</tr>
<tr>
<td>PiYG</td>
<td>pink, white, yellow-green</td>
</tr>
<tr>
<td>PRGn</td>
<td>purple, white, green</td>
</tr>
<tr>
<td>PuOr</td>
<td>orange, white, purple</td>
</tr>
<tr>
<td>RdBu</td>
<td>red, white, blue</td>
</tr>
<tr>
<td>RdGy</td>
<td>red, white, gray</td>
</tr>
<tr>
<td>RdY1Bu</td>
<td>red, yellow, blue</td>
</tr>
<tr>
<td>RdY1Gn</td>
<td>red, yellow, green</td>
</tr>
<tr>
<td>Spectral</td>
<td>red, orange, yellow, green, blue</td>
</tr>
</tbody>
</table>

ColorBrewer Sequential (luminance decreases monotonically):

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blues</td>
<td>white to dark blue</td>
</tr>
<tr>
<td>BuGn</td>
<td>white, light blue, dark green</td>
</tr>
<tr>
<td>BuPu</td>
<td>white, light blue, dark purple</td>
</tr>
<tr>
<td>GnBu</td>
<td>white, light green, dark blue</td>
</tr>
<tr>
<td>Greens</td>
<td>white to dark green</td>
</tr>
<tr>
<td>Greys</td>
<td>white to black (not linear)</td>
</tr>
<tr>
<td>Oranges</td>
<td>white, orange, dark brown</td>
</tr>
<tr>
<td>OrRd</td>
<td>white, orange, dark red</td>
</tr>
<tr>
<td>PuBu</td>
<td>white, light purple, dark blue</td>
</tr>
<tr>
<td>PuBuGn</td>
<td>white, light purple, dark green</td>
</tr>
<tr>
<td>PuRd</td>
<td>white, light purple, dark red</td>
</tr>
<tr>
<td>Purples</td>
<td>white to dark purple</td>
</tr>
<tr>
<td>RdPu</td>
<td>white, pink, dark purple</td>
</tr>
<tr>
<td>Reds</td>
<td>white to dark red</td>
</tr>
<tr>
<td>Y1Gn</td>
<td>light yellow, dark green</td>
</tr>
<tr>
<td>Y1GnBu</td>
<td>light yellow, light green, dark blue</td>
</tr>
<tr>
<td>Y1OrBr</td>
<td>light yellow, orange, dark brown</td>
</tr>
<tr>
<td>Y1OrRd</td>
<td>light yellow, orange, dark red</td>
</tr>
</tbody>
</table>

ColorBrewer Qualitative:

(For plotting nominal data, ListedColormap should be used, not LinearSegmentedColormap. Different sets of colors are recommended for different numbers of categories. These continuous versions of the qualitative schemes may be removed or converted in the future.)

- Accent
- Dark2
- Paired
- Pastel1
- Pastel2
- Set1
- Set2
- Set3

Other miscellaneous schemes:
<table>
<thead>
<tr>
<th>Colorormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>afmhot</td>
<td>sequential black-orange-yellow-white blackbody spectrum, commonly used in atomic force microscopy</td>
</tr>
<tr>
<td>brg</td>
<td>blue-red-green</td>
</tr>
<tr>
<td>bwr</td>
<td>diverging blue-white-red</td>
</tr>
<tr>
<td>coolwarm</td>
<td>diverging blue-gray-red, meant to avoid issues with 3D shading, color blindness, and ordering of colors</td>
</tr>
<tr>
<td>CM-Rmap</td>
<td>“Default colormaps on color images often reproduce to confusing grayscale images. The proposed colormap maintains an aesthetically pleasing color image that automatically reproduces to a monotonic grayscale with discrete, quantifiable saturation levels.”</td>
</tr>
<tr>
<td>cube-helix</td>
<td>Unlike most other color schemes cubehelix was designed by D.A. Green to be monotonically increasing in terms of perceived brightness. Also, when printed on a black and white postscript printer, the scheme results in a greyscale with monotonically increasing brightness. This color scheme is named cubehelix because the r,g,b values produced can be visualised as a squashed helix around the diagonal in the r,g,b color cube.</td>
</tr>
<tr>
<td>gnu-plot</td>
<td>gnuplot’s traditional pm3d scheme (black-blue-red-yellow)</td>
</tr>
<tr>
<td>gnu-plot2</td>
<td>sequential color printable as gray (black-blue-violet-yellow-white)</td>
</tr>
<tr>
<td>ocean</td>
<td>green-blue-white</td>
</tr>
<tr>
<td>rainbow</td>
<td>spectral purple-blue-green-yellow-orange-red colormap with diverging luminance</td>
</tr>
<tr>
<td>seismic</td>
<td>diverging blue-white-red</td>
</tr>
<tr>
<td>nipy_spectral</td>
<td>nipy_spectral was designed by D.A. Green to be monotonically increasing in terms of perceived brightness. Also, when printed on a black and white postscript printer, the scheme results in a greyscale with monotonically increasing brightness. This color scheme is named cubehelix because the r,g,b values produced can be visualised as a squashed helix around the diagonal in the r,g,b color cube.</td>
</tr>
<tr>
<td>terrain</td>
<td>terrain was designed by D.A. Green to be monotonically increasing in terms of perceived brightness. Also, when printed on a black and white postscript printer, the scheme results in a greyscale with monotonically increasing brightness. This color scheme is named cubehelix because the r,g,b values produced can be visualised as a squashed helix around the diagonal in the r,g,b color cube.</td>
</tr>
</tbody>
</table>

The following colormaps are redundant and may be removed in future versions. It’s recommended to use the names in the descriptions instead, which produce identical output:

<table>
<thead>
<tr>
<th>Colorormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gist_gray</td>
<td>identical to gray</td>
</tr>
<tr>
<td>gist_yarg</td>
<td>identical to gray_r</td>
</tr>
<tr>
<td>binary</td>
<td>identical to gray_r</td>
</tr>
<tr>
<td>spectral</td>
<td>identical to nipy_spectral</td>
</tr>
</tbody>
</table>

---

3See Diverging Color Maps for Scientific Visualization by Kenneth Moreland.

4See A Color Map for Effective Black-and-White Rendering of Color-Scale Images by Carey Rappaport

5Changed to distinguish from ColorBrewer’s Spectral map. spectral() still works, but set_cmap('nipy_spectral') is recommended for clarity.
This chapter is a log of changes to matplotlib that affect the outward-facing API. If updating matplotlib breaks your scripts, this list may help describe what changes may be necessary in your code or help figure out possible sources of the changes you are experiencing.

For new features that were added to matplotlib, please see *What’s new in matplotlib*.

### 34.1 Changes in 1.4.x

#### 34.1.1 Code changes

- A major refactoring of the axes module was made. The axes module has been split into smaller modules:
  - the _base module, which contains a new private _AxesBase class. This class contains all methods except plotting and labelling methods.
  - the axes module, which contains the Axes class. This class inherits from _AxesBase, and contains all plotting and labelling methods.
  - the _subplot module, with all the classes concerning subplotting.

There are a couple of things that do not exists in the axes module’s namespace anymore. If you use them, you need to import them from their original location:

- math -> import math
- ma -> from numpy import ma
- cbook -> from matplotlib import cbook
- docstring -> from matplotlib import docstring
- is_sequence_of_strings -> from matplotlib.cbook import is_sequence_of_strings
- is_string_like -> from matplotlib.cbook import is_string_like
- iterable -> from matplotlib.cbook import iterable
- itertools -> import itertools
- martist -> from matplotlib import artist as martist
- matplotlib -> import matplotlib
• mcoll -> from matplotlib import collections as mcoll
• mcolors -> from matplotlib import colors as mcolors
• mcontour -> from matplotlib import contour as mcontour
• mpatches -> from matplotlib import patches as mpatches
• mpath -> from matplotlib import path as mpath
• mquiver -> from matplotlib import quiver as mquiver
• mstack -> from matplotlib import stack as mstack
• mstream -> from matplotlib import stream as mstream
• mtable -> from matplotlib import table as mtable

As part of the refactoring to enable Qt5 support, the module matplotlib.backends.qt4_compat was renamed to matplotlib.qt_compat. qt4_compat is deprecated in 1.4 and will be removed in 1.5.

• The errorbar() method has been changed such that the upper and lower limits (lolims, uplims, xlolims, xuplims) now point in the correct direction.

• The fmt kwarg for plot() defaults.

• A bug has been fixed in the path effects rendering of fonts, which now means that the font size is consistent with non-path effect fonts. See https://github.com/matplotlib/matplotlib/issues/2889 for more detail.

• The Sphinx extensions ipython_directive and ipython_console_highlighting have been moved to the IPython project itself. While they remain in matplotlib for this release, they have been deprecated. Update your extensions in conf.py to point to IPython.sphinxext.ipython_directive instead of matplotlib.sphinxext.ipython_directive.

• In finance, almost all functions have been deprecated and replaced with a pair of functions name *_ochl and *_ohlc. The former is the ‘open-close-high-low’ order of quotes used previously in this module, and the latter is the ‘open-high-low-close’ order that is standard in finance.

• For consistency the face_alpha keyword to matplotlib.patheffects.SimplePatchShadow has been deprecated in favour of the alpha keyword. Similarly, the keyword offset_xy is now named offset across all _Base‘ has been renamed to matplotlib.patheffects.AbstractPathEffect. matplotlib.patheffect.ProxyRenderer has been renamed to matplotlib.patheffects.PathEffectRenderer and is now a full RendererBase subclass.

• The artist used to draw the outline of a colorbar has been changed from a matplotlib.lines.Line2D to matplotlib.patches.Polygon, thus colorbar.ColorbarBase.outline is now a matplotlib.patches.Polygon object.

• The legend handler interface has changed from a callable, to any object which implements the legend_artists method (a deprecation phase will see this interface be maintained for v1.4). See Legend guide for further details. Further legend changes include:

  – matplotlib.axes.Axes._get_legend_handles() now returns a generator of handles, rather than a list.
- The `legend()` function’s “loc” positional argument has been deprecated. Use the “loc” keyword instead.

- The `rcParams` savefig.transparent has been added to control default transparency when saving figures.

- Slightly refactored the `Annotation` family. The text location in `Annotation` is now handled entirely handled by the underlying `Text` object so `set_position` works as expected. The attributes `xytext` and `textcoords` have been deprecated in favor of `xyann` and `anncoords` so that `Annotation` and `AnnotationBbox` can share a common sensibly named api for getting/setting the location of the text or box.
  - `xyann` -> set the location of the annotation
  - `xy` -> set where the arrow points to
  - `anncoords` -> set the units of the annotation location
  - `xycoords` -> set the units of the point location
  - `set_position()` -> `Annotation` only set location of annotation

- `matplotlib.mlab.specgram`, `matplotlib.mlab.psd`, `matplotlib.mlab.csd`, `matplotlib.mlab.cohere`, `matplotlib.mlab.cohere_pairs`, `matplotlib.pyplot.specgram`, `matplotlib.pyplot.psd`, `matplotlib.pyplot.csd`, and `matplotlib.pyplot.cohere` now raise `ValueError` where they previously raised `AssertionError`.

- For `matplotlib.mlab.psd`, `matplotlib.mlab.csd`, `matplotlib.mlab.cohere`, `matplotlib.mlab.cohere_pairs`, `matplotlib.pyplot.specgram`, `matplotlib.pyplot.psd`, `matplotlib.pyplot.csd`, and `matplotlib.pyplot.cohere`, in cases where a shape (n, 1) array is returned, this is now converted to a (n, ) array. Previously, (n, m) arrays were averaged to an (n, ) array, but (n, 1) arrays were returend unchanged. This change makes the dimensions consistent in both cases.

- Added the `rcParam` axes.fromatter.useoffset to control the default value of `useOffset` in `ticker.ScalarFormatter`.

- Added `Formatter` sub-class `StrMethodFormatter` which does the exact same thing as `FormatStrFormatter`, but for new-style formatting strings.

- Deprecated `matplotlib.testing.image_util` and the only function within, `matplotlib.testing.image_util.autocontrast`. These will be removed completely in `v1.5.0`.

- The `fmt` argument of `plot_date()` has been changed from `bo` to just `o`, so color cycling can happen by default.

- Removed the class `FigureManagerQTAgg` and deprecated `NavigationToolbar2QTAgg` which will be removed in `1.5`.

- Removed formerly public (non-prefixed) attributes `rect` and `drawRect` from `FigureCanvasQTAgg`; they were always an implementation detail of the (preserved) `drawRectangle()` function.

- The function signatures of `tight_bbox.adjust_bbox` and `tight_bbox.process_figure_for_rasterizing` have been changed. A new `fixed_dpi` parameter allows for overriding the `figure.dpi` setting instead of trying to deduce the intended behaviour from the file format.
• Added support for horizontal/vertical axes padding to mpl_toolkits.axes_grid1.ImageGrid — argument axes_pad can now be tuple-like if separate axis padding is required. The original behavior is preserved.

• Added support for skewed transforms to matplotlib.transforms.Affine2D, which can be created using the skew and skew_deg methods.

• Added clockwise parameter to control sectors direction in axes.pie

• In matplotlib.lines.Line2D the markevery functionality has been extended. Previously an integer start-index and stride-length could be specified using either a two-element-list or a two-element-tuple. Now this can only be done using a two-element-tuple. If a two-element-list is used then it will be treated as numpy fancy indexing and only the two markers corresponding to the given indexes will be shown.

• removed prop kwarg from mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar call. It was passed through to the base-class __init__ and is only used for setting padding. Now fontproperties (which is what is really used to set the font properties of AnchoredSizeBar) is passed through in place of prop. If fontproperties is not passed in, but prop is, then prop is used inplace of fontproperties. If both are passed in, prop is silently ignored.

• The use of the index 0 in plt.subplot and related commands is deprecated. Due to a lack of validation calling plt.subplots(2, 2, 0) does not raise an exception, but puts an axes in the _last_ position. This is due to the indexing in subplot being 1-based (to mirror MATLAB) so before indexing into the GridSpec object used to determine where the axes should go, 1 is subtracted off. Passing in 0 results in passing -1 to GridSpec which results in getting the last position back. Even though this behavior is clearly wrong and not intended, we are going through a deprecation cycle in an abundance of caution that any users are exploiting this ‘feature’. The use of 0 as an index will raise a warning in 1.4 and an exception in 1.5.

• Clipping is now off by default on offset boxes.

• matplotlib now uses a less-aggressive call to gc.collect(1) when closing figures to avoid major delays with large numbers of user objects in memory.

• The default clip value of all pie artists now defaults to False.

34.1.2 Code removal

• Removed mlab.levypdf. The code raised a numpy error (and has for a long time) and was not the standard form of the Levy distribution. scipy.stats.levy should be used instead

34.2 Changes in 1.3.x

34.2.1 Changes in 1.3.1

It is rare that we make an API change in a bugfix release, however, for 1.3.1 since 1.3.0 the following change was made:

• text.Text.cached (used to cache font objects) has been made into a private variable. Among the obvious encapsulation benefit, this removes this confusing-looking member from the documentation.
• The method \texttt{hist()} now always returns bin occupancies as an array of type \texttt{float}. Previously, it was sometimes an array of type \texttt{int}, depending on the call.

34.2.2 Code removal

• The following items that were deprecated in version 1.2 or earlier have now been removed completely.

  – The Qt 3.x backends (\texttt{qt} and \texttt{qtagg}) have been removed in favor of the Qt 4.x backends (\texttt{qt4} and \texttt{qt4agg}).
  – The FltkAgg and Emf backends have been removed.
  – The \texttt{matplotlib.nxutils} module has been removed. Use the functionality on \texttt{matplotlib.path.Path.contains_point} and friends instead.
  – Instead of \texttt{axes.Axes.get_frame}, use \texttt{axes.Axes.patch}.
  – The following \texttt{kwargs} to the \texttt{legend} function have been renamed:
    * \texttt{pad} -> \texttt{borderpad}
    * \texttt{labelsep} -> \texttt{labelspacing}
    * \texttt{handlelen} -> \texttt{handlelength}
    * \texttt{handletextsep} -> \texttt{handletextpad}
    * \texttt{axespad} -> \texttt{borderaxespad}
  
  Related to this, the following rcParams have been removed:
  * \texttt{legend.pad}, \texttt{legend.labelsep}, \texttt{legend.handlelen}, \texttt{legend.handletextsep} and \texttt{legend.axespad}
  – For the \texttt{hist} function, instead of \texttt{width}, use \texttt{rwidth} (relative width).
  – On \texttt{patches.Circle}, the \texttt{resolution} kwarg has been removed. For a circle made up of line segments, use \texttt{patches.CirclePolygon}.
  – The printing functions in the Wx backend have been removed due to the burden of keeping them up-to-date.
  – \texttt{mlab.liaupunov} has been removed.
  – \texttt{mlab.save}, \texttt{mlab.load}, \texttt{pylab.save} and \texttt{pylab.load} have been removed. We recommend using \texttt{numpy.savetxt} and \texttt{numpy.loadtxt} instead.
  – \texttt{widgets.HorizontalSpanSelector} has been removed. Use \texttt{widgets.SpanSelector} instead.

34.2.3 Code deprecation

• The CocoaAgg backend has been deprecated, with the possibility for deletion or resurrection in a future release.
The top-level functions in `matplotlib.path` that are implemented in C++ were never meant to be public. Instead, users should use the Pythonic wrappers for them in the `path.Path` and `collections.Collection` classes. Use the following mapping to update your code:

- `point_in_path` -> `path.Path.contains_point`
- `get_path_extents` -> `path.Path.get_extents`
- `point_in_path_collection` -> `collection.Collection.contains`
- `path_in_path` -> `path.Path.contains_path`
- `path_intersects_path` -> `path.Path.intersects_path`
- `convert_path_to_polygons` -> `path.Path.to_polygons`
- `cleanup_path` -> `path.Path.cleaned`
- `points_in_path` -> `path.Path.contains_points`
- `clip_path_to_rect` -> `path.Path.clip_to_bbox`

- `matplotlib.colors.normalize` and `matplotlib.colors.no_norm` have been deprecated in favour of `matplotlib.colors.Normalize` and `matplotlib.colors.NoNorm` respectively.

- The `ScalarMappable` class' `set_colorbar` is now deprecated. Instead, the `matplotlib.cm.ScalarMappable.colorbar` attribute should be used. In previous `matplotlib` versions this attribute was an undocumented tuple of `(colorbar_instance, colorbar_axes)` but is now just `colorbar_instance`. To get the colorbar axes it is possible to just use the `ax` attribute on a colorbar instance.

- The `mpl` module is now deprecated. Those who relied on this module should transition to simply using `import matplotlib as mpl`.

### 34.2.4 Code changes

- `Patch` now fully supports using RGBA values for its `facecolor` and `edgecolor` attributes, which enables faces and edges to have different alpha values. If the `Patch` object's `alpha` attribute is set to anything other than `None`, that value will override any alpha-channel value in both the face and edge colors. Previously, if `Patch` had `alpha=None`, the alpha component of `edgecolor` would be applied to both the edge and face.

- The optional `isRGB` argument to `set_foreground()` (and the other `GraphicsContext` classes that descend from it) has been renamed to `isRGBA`, and should now only be set to `True` if the `fg` color argument is known to be an RGBA tuple.

- For `Patch`, the `capstyle` used is now `butt`, to be consistent with the default for most other objects, and to avoid problems with non-solid linestyle appearing solid when using a large linewidth. Previously, `Patch` used `capstyle='projecting'`.

- `Path` objects can now be marked as `readonly` by passing `readonly=True` to its constructor. The built-in path singletons, obtained through `Path.unit*` class methods return readonly paths. If you have code that modified these, you will need to make a deepcopy first, using either:
```python
import copy
path = copy.deepcopy(Path.unit_circle())

# or
path = Path.unit_circle().deepcopy()
```

Deep copying a Path always creates an editable (i.e. non-readonly) Path.

- The list at `Path.NUM_VERTICES` was replaced by a dictionary mapping Path codes to the number of expected vertices at `NUM_VERTICES_FOR_CODE`.

- To support XKCD style plots, the `matplotlib.path.cleanup_path()` method’s signature was updated to require a sketch argument. Users of `matplotlib.path.cleanup_path()` are encouraged to use the new `cleaned()` Path method.

- Data limits on a plot now start from a state of having “null” limits, rather than limits in the range (0, 1). This has an effect on artists that only control limits in one direction, such as `axvline` and `axhline`, since their limits will not longer also include the range (0, 1). This fixes some problems where the computed limits would be dependent on the order in which artists were added to the axes.

- Fixed a bug in setting the position for the right/top spine with data position type. Previously, it would draw the right or top spine at +1 data offset.

- In `FancyArrow`, the default arrow head width, `head_width`, has been made larger to produce a visible arrow head. The new value of this kwarg is `head_width = 20 * width`.

- It is now possible to provide `number of levels + 1` colors in the case of `extend='both'` for `contourf` (or just `number of levels` colors for an extend value `min` or `max`) such that the resulting colormap’s `set_under` and `set_over` are defined appropriately. Any other number of colors will continue to behave as before (if more colors are provided than levels, the colors will be unused). A similar change has been applied to `contour`, where `extend='both'` would expect `number of levels + 2` colors.

- A new keyword `extendrect` in `colorbar()` and `ColorbarBase` allows one to control the shape of colorbar extensions.

- The extension of `MultiCursor` to both vertical (default) and/or horizontal cursor implied that `self.line` is replaced by `self.vline` for vertical cursors lines and `self.hline` is added for the horizontal cursors lines.

- On POSIX platforms, the `report_memory()` function raises `NotImplementedError` instead of `OSError` if the `ps` command cannot be run.

- The `matplotlib.cbook.check_output()` function has been moved to `matplotlib.compat.subprocess()`.

### 34.2.5 Configuration and rcParams

- On Linux, the user-specific `matplotlibrc` configuration file is now located in `config/matplotlib/matplotlibrc` to conform to the XDG Base Directory Specification.
• The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.

• Removed call of `grid()` in `plotfile()`. To draw the axes grid, set the `axes.grid` rcParam to `True`, or explicitly call `grid()`.

### 34.3 Changes in 1.2.x

• The classic option of the rc parameter `toolbar` is deprecated and will be removed in the next release.

• The `isvector()` method has been removed since it is no longer functional.

• The `rasterization_zorder` property on `Axes` a zorder below which artists are rasterized. This has defaulted to `-30000.0`, but it now defaults to `None`, meaning no artists will be rasterized. In order to rasterize artists below a given zorder value, `set_rasterization_zorder` must be explicitly called.

• In `scatter()`, and `scatter`, when specifying a marker using a tuple, the angle is now specified in degrees, not radians.

• Using `twinx()` or `twiny()` no longer overrides the current locaters and formatters on the axes.

• In `contourf()`, the handling of the `extend` kwarg has changed. Formerly, the extended ranges were mapped after to 0, 1 after being normed, so that they always corresponded to the extreme values of the colormap. Now they are mapped outside this range so that they correspond to the special colormap values determined by the `set_under()` and `set_over()` methods, which default to the colormap end points.

• The new rc parameter `savefig.format` replaces `cairo.format` and `savefig.extension`, and sets the default file format used by `matplotlib.figure.Figure.savefig()`.

• In `pie()` and `pie()`, one can now set the radius of the pie; setting the `radius` to ‘None’ (the default value), will result in a pie with a radius of 1 as before.

• Use of `projection_factory()` is now deprecated in favour of axes class identification using `process_projection_requirements()` followed by direct axes class invocation (at the time of writing, functions which do this are: `add_axes()`, `add_subplot()` and `gca()`). Therefore:

```python
key = figure._make_key(*args, **kwargs)
ispolar = kwvars.pop('polar', False)
projection = kwvars.pop('projection', None)
if ispolar:
    if projection is not None and projection != 'polar':
        raise ValueError('polar and projection args are inconsistent')
    projection = 'polar'
ax = projection_factory(projection, self, rect, **kwvars)
key = self._make_key(*args, **kwvars)

# is now

projection_class, kwvars, key = \
```
process_projection_requirements(self, *args, **kwargs)
ax = projection_class(self, rect, **kwargs)

This change means that third party objects can expose themselves as matplotlib axes by providing a `_as_mpl_axes` method. See `Adding new scales and projections to matplotlib` for more detail.

• A new keyword `extendfrac` in `colorbar()` and `ColorbarBase` allows one to control the size of the triangular minimum and maximum extensions on colorbars.

• A new keyword `capthick` in `errorbar()` has been added as an intuitive alias to the `markeredgewidth` and `mew` keyword arguments, which indirectly controlled the thickness of the caps on the errorbars. For backwards compatibility, specifying either of the original keyword arguments will override any value provided by `capthick`.

• Transform subclassing behaviour is now subtly changed. If your transform implements a non-affine transformation, then it should override the `transform_non_affine` method, rather than the generic `transform` method. Previously transforms would define `transform` and then copy the method into `transform_non_affine`:

```python
class MyTransform(mtrans.Transform):
    def transform(self, xy):
        ...
    transform_non_affine = transform
```

This approach will no longer function correctly and should be changed to:

```python
class MyTransform(mtrans.Transform):
    def transform_non_affine(self, xy):
        ...
```

• Artists no longer have `x_isdata` or `y_isdata` attributes; instead any artist’s transform can be interrogated with `artist_instance.get_transform().contains_branch(ax.transData)`

• Lines added to an axes now take into account their transform when updating the data and view limits. This means transforms can now be used as a pre-transform. For instance:

```python
>>> import matplotlib.pyplot as plt
>>> import matplotlib.transforms as mtrans

>>> ax = plt.axes()
>>> ax.plot(range(10), transform=mtrans.Affine2D().scale(10) + ax.transData)
>>> print(ax.viewLim)
Bbox('array([[ 0., 0.],
          [ 90., 90.]],
          dtype=float64)')
```

• One can now easily get a transform which goes from one transform’s coordinate system to another, in an optimized way, using the new subtract method on a transform. For instance, to go from data coordinates to axes coordinates:

```python
>>> import matplotlib.pyplot as plt
>>> ax = plt.axes()
>>> data2ax = ax.transData - ax.transAxes
>>> print(ax.transData.depth, ax.transAxes.depth)
```

34.3. Changes in 1.2.x 709
for versions before 1.2 this could only be achieved in a sub-optimal way, using `ax.transData + ax.transAxes.inverted()` (depth is a new concept, but had it existed it would return 4 for this example).

- `twinx` and `twiny` now returns an instance of SubplotBase if parent axes is an instance of SubplotBase.
- All Qt3-based backends are now deprecated due to the lack of py3k bindings. Qt and QtAgg backends will continue to work in v1.2.x for py2.6 and py2.7. It is anticipated that the Qt3 support will be completely removed for the next release.
- `ColorConverter`, `Colormap` and `Normalize` now subclasses `object`.
- `ContourSet` instances no longer have a `transform` attribute. Instead, access the transform with the `get_transform` method.

### 34.4 Changes in 1.1.x

- Added new `matplotlib.sankey.Sankey` for generating Sankey diagrams.
- In `imshow()`, setting `interpolation` to ‘nearest’ will now always mean that the nearest-neighbor interpolation is performed. If you want the no-op interpolation to be performed, choose ‘none’.
- There were errors in how the tri-functions were handling input parameters that had to be fixed. If your tri-plots are not working correctly anymore, or you were working around apparent mistakes, please see issue #203 in the github tracker. When in doubt, use kwargs.
- The ‘symlog’ scale had some bad behavior in previous versions. This has now been fixed and users should now be able to use it without frustrations. The fixes did result in some minor changes in appearance for some users who may have been depending on the bad behavior.
- There is now a common set of markers for all plotting functions. Previously, some markers existed only for `scatter()` or just for `plot()`. This is now no longer the case. This merge did result in a conflict. The string ‘d’ now means “thin diamond” while ‘D’ will mean “regular diamond”.

### 34.5 Changes beyond 0.99.x

- The default behavior of `matplotlib.axes.Axes.set_xlim()`, `matplotlib.axes.Axes.set_ylim()`, and `matplotlib.axes.Axes.axis()`, and their corresponding pyplot functions, has been changed: when view limits are set explicitly with one of these methods, autoscaling is turned off for the matching axis. A new `auto` kwarg is available to control this behavior. The limit kwargs have been renamed to `left` and `right` instead of `xmin` and `xmax`, and `bottom` and `top` instead of `ymin` and `ymax`. The old names may still be used, however.
- There are five new Axes methods with corresponding pyplot functions to facilitate autoscaling, tick location, and tick label formatting, and the general appearance of ticks and tick labels:
- `matplotlib.axes.Axes.autoscale()` turns autoscaling on or off, and applies it.

- `matplotlib.axes.Axes.margins()` sets margins used to autoscale the `matplotlib.axes.Axes.viewLim` based on the `matplotlib.axes.Axes.dataLim`.

- `matplotlib.axes.Axes.locator_params()` allows one to adjust axes locator parameters such as `nbins`.

- `matplotlib.axes.Axes.ticklabel_format()` is a convenience method for controlling the `matplotlib.ticker.ScalarFormatter` that is used by default with linear axes.

- `matplotlib.axes.Axes.tick_params()` controls direction, size, visibility, and color of ticks and their labels.

- The `matplotlib.axes.Axes.bar()` method accepts a `error_kw` kwarg; it is a dictionary of kwargs to be passed to the errorbar function.

- The `matplotlib.axes.Axes.hist()` `color` kwarg now accepts a sequence of color specs to match a sequence of datasets.

- The `EllipseCollection` has been changed in two ways:
  - There is a new `units` option, `xy`, that scales the ellipse with the data units. This matches the `class:``-matplotlib.patches.Ellipse` scaling.
  - The `height` and `width` kwargs have been changed to specify the height and width, again for consistency with `Ellipse`, and to better match their names; previously they specified the half-height and half-width.

- There is a new rc parameter `axes.color_cycle`, and the color cycle is now independent of the rc parameter `lines.color`. `matplotlib.Axes.set_default_color_cycle()` is deprecated.

- You can now print several figures to one pdf file and modify the document information dictionary of a pdf file. See the docstrings of the class `matplotlib.backends.backend_pdf.PdfPages` for more information.

- Removed `configobj` and `enthought.traits` packages, which are only required by the experimental traited config and are somewhat out of date. If needed, install them independently.

- The new rc parameter `savefig.extension` sets the filename extension that is used by `matplotlib.figure.Figure.savefig()` if its `fname` argument lacks an extension.

- In an effort to simplify the backend API, all clipping rectangles and paths are now passed in using `GraphicsContext` objects, even on collections and images. Therefore:

  ```python
def draw_path_collection(self, master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds, urls)
  # is now

def draw_path_collection(self, gc, master_transform, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds, urls)
``
There are four new Axes methods with corresponding pyplot functions that deal with unstructured triangular grids:

- `matplotlib.axes.Axes.tricontour()` draws contour lines on a triangular grid.
- `matplotlib.axes.Axes.tricontourf()` draws filled contours on a triangular grid.
- `matplotlib.axes.Axes.tripcolor()` draws a pseudocolor plot on a triangular grid.
- `matplotlib.axes.Axes.triplot()` draws a triangular grid as lines and/or markers.

### 34.6 Changes in 0.99

- `pylab` no longer provides a load and save function. These are available in `matplotlib.mlab`, or you can use `numpy.loadtxt` and `numpy.savetxt` for text files, or `np.save` and `np.load` for binary `numpy` arrays.
- User-generated colormaps can now be added to the set recognized by `matplotlib.cm.get_cmap()`. Colormaps can be made the default and applied to the current image using `matplotlib.pyplot.set_cmap()`.
- Changed `use_mrecords` default to `False` in `mlab.csv2rec` since this is partially broken.
- Axes instances no longer have a “frame” attribute. Instead, use the new “spines” attribute. Spines is a dictionary where the keys are the names of the spines (e.g., ‘left’, ‘right’ and so on) and the values are the artists that draw the spines. For normal (rectilinear) axes, these artists are `Line2D` instances. For other axes (such as polar axes), these artists may be `Patch` instances.
- Polar plots no longer accept a resolution kwarg. Instead, each Path must specify its own number of interpolation steps. This is unlikely to be a user-visible change – if interpolation of data is required, that should be done before passing it to `matplotlib`.
34.7 Changes for 0.98.x

- `psd()`, `csd()`, and `cohere()` will now automatically wrap negative frequency components to the beginning of the returned arrays. This is much more sensible behavior and makes them consistent with `specgram()`. The previous behavior was more of an oversight than a design decision.
- Added new keyword parameters `nonposx`, `nonposy` to `matplotlib.axes.Axes` methods that set log scale parameters. The default is still to mask out non-positive values, but the kwargs accept ‘clip’, which causes non-positive values to be replaced with a very small positive value.
- Added new `matplotlib.pyplot.fignum_exists()` and `matplotlib.pyplot.get_fignums()`; they merely expose information that had been hidden in `matplotlib._pylab_helpers`.
- Deprecated `numerix` package.
- Added new `matplotlib.image.imsave()` and exposed it to the `matplotlib.pyplot` interface.
- Remove support for `pyExcelerator` in `exceltools` – use `xlwt` instead.
- Changed the defaults of `acorr` and `xcorr` to use `usevlines=True, maxlags=10 and normed=True` since these are the best defaults.
- Following keyword parameters for `matplotlib.label.Label` are now deprecated and new set of parameters are introduced. The new parameters are given as a fraction of the font-size. Also, `scattery-offsets, fancybox` and `columnspacing` are added as keyword parameters.

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad</td>
<td>borderpad</td>
</tr>
<tr>
<td>labelsep</td>
<td>labelspacing</td>
</tr>
<tr>
<td>handlelen</td>
<td>handlelength</td>
</tr>
<tr>
<td>handletextsep</td>
<td>handletextpad</td>
</tr>
<tr>
<td>axespad</td>
<td>borderaxespad</td>
</tr>
</tbody>
</table>

- Removed the `configobj` and experimental traits `rc` support.
- Modified `matplotlib.mlab.psd()`, `matplotlib.mlab.csd()`, `matplotlib.mlab.coherence()`, and `matplotlib.mlab.specgram()` to scale one-sided densities by a factor of 2. Also, optionally scale the densities by the sampling frequency, which gives true values of densities that can be integrated by the returned frequency values. This also gives better MATLAB compatibility. The corresponding `matplotlib.axes.Axes` methods and `matplotlib.pyplot` functions were updated as well.
- Font lookup now uses a nearest-neighbor approach rather than an exact match. Some fonts may be different in plots, but should be closer to what was requested.
- `matplotlib.afm.AFM.get_fullname()` and `matplotlib.afm.AFM.get_familyname()` no longer raise an exception if the AFM file does not specify these optional attributes, but returns a guess based on the required `FontName` attribute.
- Changed precision kwarg in `matplotlib.pyplot.spy()`; default is 0, and the string value ‘present’ is used for sparse arrays only to show filled locations.
- `matplotlib.collections.EllipseCollection` added.
• Added angles kwarg to `matplotlib.pyplot.quiver()` for more flexible specification of the arrow angles.

• Deprecated (raise `NotImplementedError`) all the mlab2 functions from `matplotlib.mlab` out of concern that some of them were not clean room implementations.

• Methods `matplotlib.collections.Collection.get_offsets()` and `matplotlib.collections.Collection.set_offsets()` added to `Collection` base class.

• `matplotlib.figure.Figure.figurePatch` renamed `matplotlib.figure.Figure.patch`; `matplotlib.axes.Axes.axesPatch` renamed `matplotlib.axes.Axes.patch`; `matplotlib.axes.Axes.axesFrame` renamed `matplotlib.axes.Axes.frame`. `matplotlib.axes.Axes.get_frame()`, which returns `matplotlib.axes.Axes.patch`, is deprecated.

• Changes in the `matplotlib.contour.ContourLabeler` attributes (`matplotlib.pyplot.clabel()` function) so that they all have a form like `.labelAttribute`. The three attributes that are most likely to be used by end users, `.cl`, `.cl_xy` and `.cl_cvalues` have been maintained for the moment (in addition to their renamed versions), but they are deprecated and will eventually be removed.

• Moved several functions in `matplotlib.mlab` and `matplotlib.cbook` into a separate module `matplotlib.numerical_methods` because they were unrelated to the initial purpose of mlab or cbook and appeared more coherent elsewhere.

### 34.8 Changes for 0.98.1

• Removed broken `matplotlib.axes3d` support and replaced it with a non-implemented error pointing to 0.91.x

### 34.9 Changes for 0.98.0

• `matplotlib.image.imread()` now no longer always returns RGBA data—if the image is luminance or RGB, it will return a MxN or MxNx3 array if possible. Also uint8 is no longer always forced to float.

• Rewrote the `matplotlib.cm.ScalarMappable` callback infrastructure to use `matplotlib.cbook.CallbackRegistry` rather than custom callback handling. Any users of `matplotlib.cm.ScalarMappable.add_observer()` of the `ScalarMappable` should use the `matplotlib.cm.ScalarMappable.callbacks CallbackRegistry` instead.

• New axes function and Axes method provide control over the plot color cycle: `matplotlib.axes.set_default_color_cycle()` and `matplotlib.axes.Axes.set_color_cycle()`.

• `matplotlib` now requires Python 2.4, so `matplotlib.cbook` will no longer provide `set`, `enumerate()`, `reversed()` or `izip()` compatibility functions.
- In Numpy 1.0, bins are specified by the left edges only. The axes method `matplotlib.axes.Axes.hist()` now uses future Numpy 1.3 semantics for histograms. Providing binedges, the last value gives the upper-right edge now, which was implicitly set to +infinity in Numpy 1.0. This also means that the last bin doesn’t contain upper outliers any more by default.

- New axes method and pyplot function, `hexbin()`, is an alternative to `scatter()` for large datasets. It makes something like a `pcolor()` of a 2-D histogram, but uses hexagonal bins.

- New kwarg, `symmetric`, in `matplotlib.ticker.MaxNLocator` allows one require an axis to be centered around zero.

- Toolkits must now be imported from `mpl_toolkits` (not `matplotlib.toolkits`)

### 34.9.1 Notes about the transforms refactoring

A major new feature of the 0.98 series is a more flexible and extensible transformation infrastructure, written in Python/Numpy rather than a custom C extension.

The primary goal of this refactoring was to make it easier to extend matplotlib to support new kinds of projections. This is mostly an internal improvement, and the possible user-visible changes it allows are yet to come.

See `matplotlib.transforms` for a description of the design of the new transformation framework.

For efficiency, many of these functions return views into Numpy arrays. This means that if you hold on to a reference to them, their contents may change. If you want to store a snapshot of their current values, use the Numpy array method `copy()`.

The view intervals are now stored only in one place – in the `matplotlib.axes.Axes` instance, not in the locator instances as well. This means locators must get their limits from their `matplotlib.axis.Axis`, which in turn looks up its limits from the `Axes`. If a locator is used temporarily and not assigned to an Axis or Axes, (e.g., in `matplotlib.contour`), a dummy axis must be created to store its bounds. Call `matplotlib.ticker.Locator.create_dummy_axis()` to do so.

The functionality of `Pbox` has been merged with `Bbox`. Its methods now all return copies rather than modifying in place.

The following lists many of the simple changes necessary to update code from the old transformation framework to the new one. In particular, methods that return a copy are named with a verb in the past tense, whereas methods that alter an object in place are named with a verb in the present tense.
### matplotlib.transforms

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bbox.get_bounds()</td>
<td>transforms.Bbox.get_bounds</td>
</tr>
<tr>
<td>Bbox.width()</td>
<td>transforms.Bbox.width</td>
</tr>
<tr>
<td>Bbox.height()</td>
<td>transforms.Bbox.height</td>
</tr>
<tr>
<td>Bbox.intervalx().get_bounds()</td>
<td>transforms.Bbox.intervalx.getbounds</td>
</tr>
<tr>
<td>Bbox.intervalx().set</td>
<td>transforms.Bbox.intervalx.set</td>
</tr>
<tr>
<td>Bbox.intervaly().get_bounds()</td>
<td>transforms.Bbox.intervaly.getbounds</td>
</tr>
<tr>
<td>Bbox.intervaly().set</td>
<td>transforms.Bbox.intervaly.set</td>
</tr>
<tr>
<td>Bbox.xmin()</td>
<td>transforms.Bbox.xmin or transforms.Bbox.xmin</td>
</tr>
<tr>
<td>Bbox.ymin()</td>
<td>transforms.Bbox.ymin or transforms.Bbox.ymin</td>
</tr>
<tr>
<td>Bbox.xmax()</td>
<td>transforms.Bbox.xmax or transforms.Bbox.xmax</td>
</tr>
<tr>
<td>Bbox.ymax()</td>
<td>transforms.Bbox.ymax or transforms.Bbox.ymax</td>
</tr>
<tr>
<td>Bbox.overlaps(bboxes)</td>
<td>Bbox.union(bboxes) [transforms.Bbox.union() is a staticmethod.]</td>
</tr>
<tr>
<td>lbwh_to_bbox(l, b, w, h)</td>
<td>Bbox.from_bounds(x0, y0, w, h)</td>
</tr>
<tr>
<td>inverse_transform_bbox()</td>
<td>Bbox inverse_transformed(trans)</td>
</tr>
<tr>
<td>Interval.contains_open(tuple, v)</td>
<td>Interval.contains_open(tuple, v)</td>
</tr>
<tr>
<td>identity_transform()</td>
<td>matplotlib.transforms.IdentityTransform</td>
</tr>
<tr>
<td>blend_xy_sep_transform(xtrans, ytrans)</td>
<td>blended_transform_factory(xtrans, ytrans)</td>
</tr>
<tr>
<td>scale_transform(xs, ys)</td>
<td>Affine2D().scale(xs, ys)</td>
</tr>
<tr>
<td>get_bbox_transform(boxin, boxout)</td>
<td>BboxTransform(boxin, boxout) or BboxTransformFrom(boxin) or BboxTransformTo(boxout)</td>
</tr>
<tr>
<td>Transform.seq_xy_tup(points)</td>
<td>transform.transform(points)</td>
</tr>
<tr>
<td>Transform.inverse_xy_tup(points)</td>
<td>transform.inverted().transform(points)</td>
</tr>
</tbody>
</table>

### matplotlib.axes

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.get_position()</td>
<td>matplotlib.axes.Axes.get_position()</td>
</tr>
<tr>
<td>Axes.set_position()</td>
<td>matplotlib.axes.Axes.set_position()</td>
</tr>
<tr>
<td>Axes.toggle_log_linearity()</td>
<td>matplotlib.axes.Axes.toggle_log_linearity()</td>
</tr>
<tr>
<td>Subplot class</td>
<td>removed.</td>
</tr>
</tbody>
</table>

The Polar class has moved to matplotlib.projections.polar.

---

1. The Bbox is bound by the points (x0, y0) to (x1, y1) and there is no defined order to these points, that is, x0 is not necessarily the left edge of the box. To get the left edge of the Bbox, use the read-only property xmin.

2. matplotlib.axes.Axes.get_position() used to return a list of points, now it returns a matplotlib.transforms.Bbox instance.

3. matplotlib.axes.Axes.set_position() now accepts either four scalars or a matplotlib.transforms.Bbox instance.

4. Since the refactoring allows for more than two scale types ('log' or 'linear'), it no longer makes sense to have a toggle. Axes.toggle_log_linearity() has been removed.
### matplotlib.artist

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.set_clip_path(path)</td>
<td>Artist.set_clip_path(path, transform)</td>
</tr>
</tbody>
</table>

### matplotlib.collections

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle</td>
<td>linestyles</td>
</tr>
</tbody>
</table>

### matplotlib.colors

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorConvertor.to_rgba_list(c)</td>
<td>ColorConvertor.to_rgba_array(c)</td>
</tr>
<tr>
<td>[matplotlib.colors.ColorConvertor.to_rgba_array() returns an Nx4 Numpy array of RGBA color quadruples.]</td>
<td></td>
</tr>
</tbody>
</table>

### matplotlib.contour

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contour._segments</td>
<td>matplotlib.contour.Contour.get_paths'()</td>
</tr>
<tr>
<td>[Returns a list of matplotlib.path.Path instances.]</td>
<td></td>
</tr>
</tbody>
</table>

### matplotlib.figure

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure.dpi.get() / Figure.dpi.set()</td>
<td>matplotlib.figure.Figure.dpi (a property)</td>
</tr>
</tbody>
</table>

### matplotlib.patches

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch.get_verts()</td>
<td>matplotlib.patches.Patch.get_path()</td>
</tr>
<tr>
<td>[Returns a matplotlib.path.Path instance]</td>
<td></td>
</tr>
</tbody>
</table>

---

5. `matplotlib.artist.Artist.set_clip_path()` now accepts a `matplotlib.path.Path` instance and a `matplotlib.transforms.Transform` that will be applied to the path immediately before clipping.

6. Linestyles are now treated like all other collection attributes, i.e. a single value or multiple values may be provided.
**Matplotlib, Release 1.4.3**

**matplotlib.backend_bases**

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphicsContext.set_clip_rectangle(tuple)</td>
<td>GraphicsContext.set_clip_rectangle(bbox)</td>
</tr>
<tr>
<td>GraphicsContext.get_clip_path()</td>
<td>GraphicsContext.get_clip_path() 7</td>
</tr>
<tr>
<td>GraphicsContext.set_clip_path()</td>
<td>GraphicsContext.set_clip_path() 8</td>
</tr>
</tbody>
</table>

**RendererBase**

New methods:

- `draw_path(self, gc, path, transform, rgbFace)`
- `draw_markers(self, gc, marker_path, marker_trans, path, trans, rgbFace)`
  `<matplotlib.backend_bases.RendererBase.draw_markers()`
- `draw_path_collection(self, master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewights, linestyles, antialiaseds) [optional]`

Changed methods:

- `draw_image(self, x, y, im, bbox)` is now `draw_image(self, x, y, im, bbox, clippath, clippath_trans)`

Removed methods:

- `draw_arc`
- `draw_line_collection`
- `draw_line`
- `draw_lines`
- `draw_point`
- `draw_quad_mesh`
- `draw_poly_collection`
- `draw_polygon`
- `draw_rectangle`
- `draw_regpoly_collection`

---

7 `matplotlib.backend_bases.GraphicsContext.get_clip_path()` returns a tuple of the form `(path, affine_transform)`, where `path` is a `matplotlib.path.Path` instance and `affine_transform` is a `matplotlib.transforms.Affine2D` instance.

8 `matplotlib.backend_bases.GraphicsContext.set_clip_path()` now only accepts a `matplotlib.transforms.TransformedPath` instance.
34.10 Changes for 0.91.2

- For csv2rec(), checkrows=0 is the new default indicating all rows will be checked for type inference
- A warning is issued when an image is drawn on log-scaled axes, since it will not log-scale the image data.
- Moved rec2gtk() to matplotlib.toolkits.gtktools
- Moved rec2excel() to matplotlib.toolkits.exceltools
- Removed, dead/experimental ExampleInfo, Namespace and Importer code from matplotlib.__init__

34.11 Changes for 0.91.1

34.12 Changes for 0.91.0

- Changed cbook.is_file_like() to cbook.is_writable_file_like() and corrected behavior.
- Added ax kwarg to pyplot.colorbar() and Figure.colorbar() so that one can specify the axes object from which space for the colorbar is to be taken, if one does not want to make the colorbar axes manually.
- Changed cbook.reversed() so it yields a tuple rather than a (index, tuple). This agrees with the python reversed builtin, and cbook only defines reversed if python doesn’t provide the builtin.
- Made skiprows=1 the default on csv2rec()
- The gd and paint backends have been deleted.
- The errorbar method and function now accept additional kwargs so that upper and lower limits can be indicated by capping the bar with a caret instead of a straight line segment.
- The matplotlib.dviread file now has a parser for files like psfonts.map and pdftex.map, to map TeX font names to external files.
- The file matplotlib.type1font contains a new class for Type 1 fonts. Currently it simply reads pfa and pfb format files and stores the data in a way that is suitable for embedding in pdf files. In the future the class might actually parse the font to allow e.g., subsetting.
- matplotlib.FT2Font now supports FT_Attach_File(). In practice this can be used to read an afm file in addition to a pfa/pfb file, to get metrics and kerning information for a Type 1 font.
- The AFM class now supports querying CapHeight and stem widths. The get_name_char method now has an isord kwarg like get_width_char.
- Changed pcolor() default to shading='flat'; but as noted now in the docstring, it is preferable to simply use the edgecolor kwarg.
- The mathtext font commands ($\cal, \rm, \it, \tt$) now behave as TeX does: they are in effect until the next font change command or the end of the grouping. Therefore uses of $\cal{R}$ should be
Matplotlib, Release 1.4.3

changed to \$\{\text{cal R}\}\$. Alternatively, you may use the new \LaTeX\-style font commands (\texttt{\textbackslash mathcal}, \texttt{\textbackslash mathrm}, \texttt{\textbackslash mathit}, \texttt{\textbackslash mathtt}) which do affect the following group, e.g., \$\text{mathcal}\{R\}\$.

- Text creation commands have a new default linespacing and a new \texttt{linespacing} kwarg, which is a multiple of the maximum vertical extent of a line of ordinary text. The default is 1.2; \texttt{linespacing=2} would be like ordinary double spacing, for example.

- Changed default kwarg in \texttt{matplotlib.colors.Normalize.__init__}() to \texttt{clip=False}; clipping silently defeats the purpose of the special over, under, and bad values in the colormap, thereby leading to unexpected behavior. The new default should reduce such surprises.

- Made the emit property of \texttt{set_xlim()} and \texttt{set_ylim()} True by default; removed the Axes custom callback handling into a ‘callbacks’ attribute which is a \texttt{CallbackRegistry} instance. This now supports the ‘xlim_changed’ and ‘ylim_changed’ Axes events.

34.13 Changes for 0.90.1

The file \texttt{dviread.py} has a (very limited and fragile) dvi reader for \texttt{usetex} support. The API might change in the future so don’t depend on it yet.

Removed deprecated support for a float value as a gray-scale; now it must be a string, like ‘0.5’. Added alpha kwarg to \texttt{ColorConverter.to_rgba_list}.

New method \texttt{set_bounds(vmin, vmax)} for formatters, locators sets the viewInterval and dataInterval from floats.

Removed deprecated \texttt{colorbar_classic}.

\texttt{Line2D.get_xdata} and \texttt{get_ydata} valid\_only=False kwarg is replaced by \texttt{orig=True}. When True, it returns the original data, otherwise the processed data (masked, converted)

Some modifications to the \texttt{units} interface.
\texttt{units.ConversionInterface.tickers} renamed to \texttt{units.ConversionInterface.axisinfo} and it now returns a \texttt{units.AxisInfo} object rather than a tuple. This will make it easier to add axis info functionality (e.g., I added a default label on this iteration) w/o having to change the tuple length and hence the API of the client code every time new functionality is added. Also, \texttt{units.ConversionInterface.convert\_to\_value} is now simply named \texttt{units.ConversionInterface.convert}.

\texttt{Axes.errorbar} uses \texttt{Axes.vlines} and \texttt{Axes.hlines} to draw its error limits in the vertical and horizontal direction. As you’ll see in the changes below, these functions now return a \texttt{LineCollection} rather than a list of lines. The new return signature for \texttt{errorbar} is \texttt{ylins, caplines, errorcollections} where \texttt{errorcollections} is a \texttt{xerrcollection}, \texttt{yerrcollection}
Axes.vlines and Axes.hlines now create and returns a LineCollection, not a list of lines. This is much faster. The kwarg signature has changed, so consult the docs.

MaxNLocator accepts a new Boolean kwarg ('integer') to force ticks to integer locations.

Commands that pass an argument to the Text constructor or to Text.set_text() now accept any object that can be converted with '%%'. This affects xlabel(), title(), etc.

Barh now takes a **kwargs dict instead of most of the old arguments. This helps ensure that bar and barh are kept in sync, but as a side effect you can no longer pass e.g., color as a positional argument.

ft2font.get_charmap() now returns a dict that maps character codes to glyph indices (until now it was reversed)

Moved data files into lib/matplotlib so that setuptools' develop mode works. Re-organized the mpl-data layout so that this source structure is maintained in the installation. (i.e., the 'fonts' and 'images' sub-directories are maintained in site-packages.). Suggest removing site-packages/matplotlib/mpl-data and ~/.matplotlib/ttffont.cache before installing

### 34.14 Changes for 0.90.0

All artists now implement a "pick" method which users should not call. Rather, set the "picker" property of any artist you want to pick on (the epsilon distance in points for a hit test) and register with the "pick_event" callback. See examples/pick_event_demo.py for details

Bar, barh, and hist have "log" binary kwarg: log=True sets the ordinate to a log scale.

Boxplot can handle a list of vectors instead of just an array, so vectors can have different lengths.

Plot can handle 2-D x and/or y; it plots the columns.

Added linewidth kwarg to bar and barh.

Made the default Artist._transform None (rather than invoking identity_transform for each artist only to have it overridden later). Use artist.get_transform() rather than artist._transform, even in derived classes, so that the default transform will be created lazily as needed

New LogNorm subclass of Normalize added to colors.py.
All Normalize subclasses have new inverse() method, and the __call__() method has a new clip kwarg.

Changed class names in colors.py to match convention: normalize -> Normalize, no_norm -> NoNorm. Old names are still available for now.

Removed obsolete pcolor_classic command and method.

Removed lineprops and markerprops from the Annotation code and replaced them with an arrow configurable with kwarg arrowprops. See examples/annotation_demo.py - JDH

### 34.15 Changes for 0.87.7

Completely reworked the annotations API because I found the old API cumbersome. The new design is much more legible and easy to read. See matplotlib.text.Annotation and examples/annotation_demo.py

markeredgecolor and markerfacecolor cannot be configured in matplotlibrc any more. Instead, markers are generally colored automatically based on the color of the line, unless marker colors are explicitly set as kwargs - NN

Changed default comment character for load to '#'.

math_parse_s_ft2font_svg from mathtext.py & mathtext2.py now returns width, height, svg_elements. svg_elements is an instance of Bunch (cmbook.py) and has the attributes svg_glyphs and svg_lines, which are both lists.

Renderer.draw_arc now takes an additional parameter, rotation. It specifies to draw the artist rotated in degrees anti-clockwise. It was added for rotated ellipses.

Renamed Figure.set_figsizes_inches to Figure.set_size_inches to better match the get method, Figure.get_size_inches.

Removed the copy_bbox_transform from transforms.py; added shallowcopy methods to all transforms. All transforms already had deepcopy methods.

FigureManager.resize(width, height): resize the window specified in pixels

barh: x and y args have been renamed to width and bottom respectively, and their order has been swapped to maintain a (position, value) order.

bar and barh: now accept kwarg 'edgecolor'.

bar and barh: The left, height, width and bottom args can now all be scalars or sequences; see docstring.

barh: now defaults to edge aligned instead of center aligned bars

bar, barh and hist: Added a keyword arg 'align' that controls between edge or center bar alignment.

Collections: PolyCollection and LineCollection now accept vertices or segments either in the original form [(x,y), (x,y), ...] or as a 2D numerix array, with X as the first column and Y as the second. Contour and quiver output the numerix form. The transforms methods Bbox.update() and Transformation.seq_xy_tups() now accept either form.

Collections: LineCollection is now a ScalarMappable like PolyCollection, etc.

Specifying a grayscale color as a float is deprecated; use a string instead, e.g., 0.75 -> '0.75'.

Collections: initializers now accept any mpl color arg, or sequence of such args; previously only a sequence of rgba tuples was accepted.

Colorbar: completely new version and api; see docstring. The original version is still accessible as colorbar_classic, but is deprecated.

Contourf: "extend" kwarg replaces "clip_ends"; see docstring. Masked array support added to pcolormesh.

Modified aspect-ratio handling:
- Removed aspect kwarg from imshow
- Axes methods:
  - set_aspect(self, aspect, adjustable=None, anchor=None)
  - set_adjustable(self, adjustable)
  - set_anchor(self, anchor)
- Pylab interface:
  - axis('image')

Backend developers: ft2font's load_char now takes a flags argument, which you can OR together from the LOAD_XXX constants.

34.16 Changes for 0.86

Matplotlib data is installed into the matplotlib module. This is similar to package_data. This should get rid of
having to check for many possibilities in _get_data_path(). The MATPLOTLIBDATA env key is still checked first to allow for flexibility.

1) Separated the color table data from cm.py out into a new file, _cm.py, to make it easier to find the actual code in cm.py and to add new colormaps. Everything from _cm.py is imported by cm.py, so the split should be transparent.

2) Enabled automatic generation of a colormap from a list of colors in contour; see modified examples/contour_demo.py.

3) Support for imshow of a masked array, with the ability to specify colors (or no color at all) for masked regions, and for regions that are above or below the normally mapped region. See examples/image_masked.py.

4) In support of the above, added two new classes, ListedColormap, and no_norm, to colors.py, and modified the Colormap class to include common functionality. Added a clip kwarg to the normalize class.

34.17 Changes for 0.85

Made xtick and ytick separate props in rc

made pos=None the default for tick formatters rather than 0 to indicate "not supplied"

Removed "feature" of minor ticks which prevents them from overlapping major ticks. Often you want major and minor ticks at the same place, and can offset the major ticks with the pad. This could be made configurable

Changed the internal structure of contour.py to a more OO style. Calls to contour or contourf in axes.py or pylab.py now return a ContourSet object which contains references to the LineCollections or PolyCollections created by the call, as well as the configuration variables that were used. The ContourSet object is a "mappable" if a colormap was used.

Added a clip_ends kwarg to contourf. From the docstring:

* clip_ends = True
  If False, the limits for color scaling are set to the minimum and maximum contour levels.
  True (default) clips the scaling limits. Example:
  if the contour boundaries are V = [-100, 2, 1, 0, 1, 2, 100], then the scaling limits will be [-100, 100] if clip_ends is False, and [-3, 3] if clip_ends is True.

Added kwargs linewidths, antialiased, and nchunk to contourf. These are experimental; see the docstring.
Changed Figure.colorbar():
    kw argument order changed;
    if mappable arg is a non-filled ContourSet, colorbar() shows
    lines instead of polygons.
    if mappable arg is a filled ContourSet with clip_ends=True,
    the endpoints are not labelled, so as to give the
    correct impression of open-endedness.

Changed LineCollection.get_linewidths to get_linewidth, for
consistency.

34.18 Changes for 0.84

Unified argument handling between hlines and vlines. Both now
take optionally a fmt argument (as in plot) and a keyword args
that can be passed onto Line2D.

Removed all references to "data clipping" in rc and lines.py since
these were not used and not optimized. I'm sure they'll be
resurrected later with a better implementation when needed.

'set' removed - no more deprecation warnings. Use 'setp' instead.

Backend developers: Added flipud method to image and removed it
from to_str. Removed origin kwarg from backend.draw_image.
origin is handled entirely by the frontend now.

34.19 Changes for 0.83

- Made HOME/.matplotlib the new config dir where the matplotlibrc
  file, the ttf.cache, and the tex.cache live. The new default
  filenames in .matplotlib have no leading dot and are not hidden.
  e.g., the new names are matplotlibrc, tex.cache, and ttffont.cache.
  This is how ipython does it so it must be right.

If old files are found, a warning is issued and they are moved to
the new location.

- backends/__init__.py no longer imports new_figure_manager,
draw_if_interactive and show from the default backend, but puts
these imports into a call to pylab_setup. Also, the Toolbar is no
longer imported from WX/WXAgg. New usage:

        from backends import pylab_setup
        new_figure_manager, draw_if_interactive, show = pylab_setup()

- Moved Figure.get_width_height() to FigureCanvasBase. It now
  returns int instead of float.
34.20 Changes for 0.82

- toolbar import change in GTKAgg, GTKCairo and WXAgg

- Added subplot config tool to GTK* backends -- note you must now import the NavigationToolbar2 from your backend of choice rather than from backend_gtk because it needs to know about the backend specific canvas -- see examples/embedding_in_gtk2.py. Ditto for wx backend -- see examples/embedding_in_wxagg.py

- hist bin change

  Sean Richards notes there was a problem in the way we created the binning for histogram, which made the last bin underrepresented. From his post:

  I see that hist uses the linspace function to create the bins and then uses searchsorted to put the values in their correct bin. That's all good but I am confused over the use of linspace for the bin creation. I wouldn't have thought that it does what is needed, to quote the docstring it creates a "Linear spaced array from min to max". For it to work correctly shouldn't the values in the bins array be the same bound for each bin? (i.e. each value should be the lower bound of a bin). To provide the correct bins for hist would it not be something like

  ```python
  def bins(xmin, xmax, N):
      if N==1: return xmax
      dx = (xmax-xmin)/N # instead of N-1
      return xmin + dx*arange(N)
  ```

  This suggestion is implemented in 0.81. My test script with these changes does not reveal any bias in the binning

  ```python
  from matplotlib.numerix.mlab import randn, rand, zeros, Float
  from matplotlib.mlab import hist, mean

  Nbins = 50
  Ntests = 200
  results = zeros((Ntests,Nbins), typecode=Float)
  for i in range(Ntests):
    print 'computing', i
    x = rand(10000)
    n, bins = hist(x, Nbins)
    results[i] = n
  print mean(results)
  ```
34.21 Changes for 0.81

- **pylab** and **artist** "set" functions renamed to **setp** to avoid clash with python2.4 built-in set. Current version will issue a deprecation warning which will be removed in future versions.

- **imshow** interpolation arguments changes for advanced interpolation schemes. See help imshow, particularly the interpolation, filternorm and filterrad kwargs.

- Support for masked arrays has been added to the plot command and to the Line2D object. Only the valid points are plotted. A "valid_only" kwarg was added to the get_xdata() and get_ydata() methods of Line2D; by default it is False, so that the original data arrays are returned. Setting it to True returns the plottable points.

- **contour** changes:

  Masked arrays: **contour** and **contourf** now accept masked arrays as the variable to be contoured. Masking works correctly for contour, but a bug remains to be fixed before it will work for contourf. The "badmask" kwarg has been removed from both functions.

  **Level argument changes:**

  Old version: a list of levels as one of the positional arguments specified the lower bound of each filled region; the upper bound of the last region was taken as a very large number. Hence, it was not possible to specify that z values between 0 and 1, for example, be filled, and that values outside that range remain unfilled.

  New version: a list of N levels is taken as specifying the boundaries of N-1 z ranges. Now the user has more control over what is colored and what is not. Repeated calls to contourf (with different colormaps or color specifications, for example) can be used to color different ranges of z. Values of z outside an expected range are left uncolored.

  **Example:**

  Old: contourf(z, [0, 1, 2]) would yield 3 regions: 0-1, 1-2, and >2.
  New: it would yield 2 regions: 0-1, 1-2. If the same 3 regions were desired, the equivalent list of levels would be [0, 1, 2, 1e38].
34.22 Changes for 0.80

- xlim/ylim/axis always return the new limits regardless of arguments. They now take kwargs which allow you to selectively change the upper or lower limits while leaving unnamed limits unchanged. See help(xlim) for example

34.23 Changes for 0.73

- Removed deprecated ColormapJet and friends
- Removed all error handling from the verbose object
- figure num of zero is now allowed

34.24 Changes for 0.72

- Line2D, Text, and Patch copy_properties renamed update_from and moved into artist base class
- LineCollectitons.color renamed to LineCollections.set_color for consistency with set/get introspection mechanism,
- pylab figure now defaults to num=None, which creates a new figure with a guaranteed unique number
- contour method syntax changed - now it is MATLAB compatible
  unchanged: contour(Z)
  old: contour(Z, x=Y, y=Y)
  new: contour(X, Y, Z)
  see http://matplotlib.sf.net/matplotlib.pylab.html#-contour
- Increased the default resolution for save command.
- Renamed the base attribute of the ticker classes to _base to avoid conflict with the base method. Sitt for subs
- subs=none now does autosubbing in the tick locator.
- New subplots that overlap old will delete the old axes. If you do not want this behavior, use fig.add_subplot or the axes command
34.25 Changes for 0.71

Significant numerix namespace changes, introduced to resolve namespace clashes between python built-ins and mlab names. Refactored numerix to maintain separate modules, rather than folding all these names into a single namespace. See the following mailing list threads for more information and background:


OLD usage

    from matplotlib.numerix import array, mean, fft

NEW usage

    from matplotlib.numerix import array
    from matplotlib.numerix.mlab import mean
    from matplotlib.numerix.fft import fft

numerix dir structure mirrors numarray (though it is an incomplete implementation)

    numerix
    numerix/mlab
    numerix/linear_algebra
    numerix/fft
    numerix/random_array

but of course you can use 'numerix : Numeric' and still get the symbols.

pylab still imports most of the symbols from Numerix, MLab, fft, etc, but is more cautious. For names that clash with python names (min, max, sum), pylab keeps the builtins and provides the numeric versions with an a* prefix, e.g., (amin, amax, asum)

34.26 Changes for 0.70

MplEvent factored into a base class Event and derived classes MouseEvent and KeyEvent

Removed distinct set_measurement in wx toolbar
34.27 Changes for 0.65.1

removed add_axes and add_subplot from backend.bases. Use figure.add_axes and add_subplot instead. The figure now manages the current axes with gca and sca for get and set current axes. If you have code you are porting which called, e.g., figmanager.add_axes, you can now simply do figmanager.canvas.figure.add_axes.

34.28 Changes for 0.65

mpl_connect and mpl_disconnect in the MATLAB interface renamed to connect and disconnect

Did away with the text methods for angle since they were ambiguous. fontangle could mean fontstyle (oblique, etc) or the rotation of the text. Use style and rotation instead.

34.29 Changes for 0.63

Dates are now represented internally as float days since 0001-01-01, UTC.

All date tickers and formaters are now in matplotlib.dates, rather than matplotlib.tickers

converters have been abolished from all functions and classes. num2date and date2num are now the converter functions for all date plots

Most of the date tick locators have a different meaning in their constructors. In the prior implementation, the first argument was a base and multiples of the base were ticked. e.g.,

    HourLocator(5) # old: tick every 5 minutes

In the new implementation, the explicit points you want to tick are provided as a number or sequence

    HourLocator(range(0,5,61)) # new: tick every 5 minutes

This gives much greater flexibility. I have tried to make the default constructors (no args) behave similarly, where possible.

Note that YearLocator still works under the base/multiple scheme. The difference between the YearLocator and the other locators is that years are not recurrent.
Financial functions:

matplotlib.finance.quotes_historical_yahoo(ticker, date1, date2)

date1, date2 are now datetime instances. Return value is a list of quotes where the quote time is a float - days since gregorian start, as returned by date2num

See examples/finance_demo.py for example usage of new API

34.30 Changes for 0.61

canvas.connect is now deprecated for event handling. Use mpl_connect and mpl_disconnect instead. The callback signature is func(event) rather than func(widget, event)

34.31 Changes for 0.60

ColormapJet and Grayscale are deprecated. For backwards compatibility, they can be obtained either by doing

    from matplotlib.cm import ColormapJet

or

    from matplotlib.matlab import *

They are replaced by cm.jet and cm.grey

34.32 Changes for 0.54.3

removed the set_default_font / get_default_font scheme from the font_manager to unify customization of font defaults with the rest of the rc scheme. See examples/font_properties_demo.py and help(rc) in matplotlib.matlab.
34.33 Changes for 0.54

34.33.1 MATLAB interface

dpi

Several of the backends used a PIXELS_PER_INCH hack that I added to try and make images render consistently across backends. This just complicated matters. So you may find that some font sizes and line widths appear different than before. Apologies for the inconvenience. You should set the dpi to an accurate value for your screen to get true sizes.

pcolor and scatter

There are two changes to the MATLAB interface API, both involving the patch drawing commands. For efficiency, pcolor and scatter have been rewritten to use polygon collections, which are a new set of objects from matplotlib.collections designed to enable efficient handling of large collections of objects. These new collections make it possible to build large scatter plots or pcolor plots with no loops at the python level, and are significantly faster than their predecessors. The original pcolor and scatter functions are retained as pcolor_classic and scatter_classic.

The return value from pcolor is a PolyCollection. Most of the properties that are available on rectangles or other patches are also available on PolyCollections, e.g., you can say:

```
c = scatter(blah, blah)
c.set_linewidth(1.0)
c.set_facecolor('r')
c.set_alpha(0.5)
```

or:

```
c = scatter(blah, blah)
set(c, 'linewidth', 1.0, 'facecolor', 'r', 'alpha', 0.5)
```

Because the collection is a single object, you no longer need to loop over the return value of scatter or pcolor to set properties for the entire list.

If you want the different elements of a collection to vary on a property, e.g., to have different line widths, see matplotlib.collections for a discussion on how to set the properties as a sequence.

For scatter, the size argument is now in points^2 (the area of the symbol in points) as in MATLAB and is not in data coords as before. Using sizes in data coords caused several problems. So you will need to adjust your size arguments accordingly or use scatter_classic.

mathtext spacing

For reasons not clear to me (and which I’ll eventually fix) spacing no longer works in font groups. However, I added three new spacing commands which compensate for this ‘ ’ (regular space), ‘/’ (small space) and ‘hspace{frac}’ where frac is a fraction of fontsize in points. You will need to quote spaces in font strings, is:
34.33.2 Object interface - Application programmers

Autoscaling

The x and y axis instances no longer have autoscale view. These are handled by
axes.autoscale_view

Axes creation

You should not instantiate your own Axes any more using the OO API. Rather, create a Figure as
before and in place of:

```python
f = Figure(figsize=(5,4), dpi=100)
a = Subplot(f, 111)
f.add_axis(a)
```

use:

```python
f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
```

That is, add_axis no longer exists and is replaced by:

```python
add_axes(rect, axisbg=defaultcolor, frameon=True)
add_subplot(num, axisbg=defaultcolor, frameon=True)
```

Artist methods

If you define your own Artists, you need to rename the _draw method to draw

Bounding boxes

matplotlib.transforms.Bound2D is replaced by matplotlib.transforms.Bbox. If you want to
construct a bbox from left, bottom, width, height (the signature for Bound2D), use matplotlib.transforms.lbwh_to_bbox, as in

```python
bbox = clickBBox = lbwh_to_bbox(left, bottom, width, height)
```

The Bbox has a different API than the Bound2D. e.g., if you want to get the width and height of
the bbox

```python
OLD::  width = fig.bbox.x.interval() height = fig.bbox.y.interval()
New::  width = fig.bbox.width() height = fig.bbox.height()
```
Object constructors

You no longer pass the bbox, dpi, or transforms to the various Artist constructors. The old way of creating lines and rectangles was cumbersome because you had to pass so many attributes to the Line2D and Rectangle classes not related directly to the geometry and properties of the object. Now default values are added to the object when you call axes.add_line or axes.add_patch, so they are hidden from the user.

If you want to define a custom transformation on these objects, call o.set_transform(trans) where trans is a Transformation instance.

In prior versions of you wanted to add a custom line in data coords, you would have to do

```python
l = Line2D(dpi, bbox, x, y, color=color, transx=transx, transy=transy, )
```

now all you need is

```python
l = Line2D(x, y, color=color)
```

and the axes will set the transformation for you (unless you have set your own already, in which case it will eave it unchanged)

Transformations

The entire transformation architecture has been rewritten. Previously the x and y transformations where stored in the xaxis and yaxis instances. The problem with this approach is it only allows for separable transforms (where the x and y transformations don’t depend on one another). But for cases like polar, they do. Now transformations operate on x,y together. There is a new base class matplotlib.transforms.Transformation and two concrete implementations, matplotlib.transforms.SeparableTransformation and matplotlib.transforms.Affine. The Separable-Transformation is constructed with the bounding box of the input (this determines the rectangular coordinate system of the input, i.e., the x and y view limits), the bounding box of the display, and possibly nonlinear transformations of x and y. The 2 most frequently used transformations, data coordinates -> display and axes coordinates -> display are available as ax.transData and ax.transAxes. See alignment_demo.py which uses axes coords.

Also, the transformations should be much faster now, for two reasons

- they are written entirely in extension code
- because they operate on x and y together, they can do the entire transformation in one loop.

Earlier I did something along the lines of:

```python
xt = sx*func(x) + tx
yt = sy*func(y) + ty
```

Although this was done in numerix, it still involves 6 length(x) for-loops (the multiply, add, and function evaluation each for x and y). Now all of that is done in a single pass.

If you are using transformations and bounding boxes to get the cursor position in data coordinates, the method calls are a little different now. See the updated examples/coords_demo.py which shows you how to do this.
Likewise, if you are using the artist bounding boxes to pick items on the canvas with the GUI, the bbox methods are somewhat different. You will need to see the updated examples/object_picker.py.

See unit/transforms_unit.py for many examples using the new transformations.

### 34.34 Changes for 0.50

* refactored Figure class so it is no longer backend dependent. FigureCanvasBackend takes over the backend specific duties of the Figure. matplotlib.backend_bases.FigureBase moved to matplotlib.figure.Figure.

* backends must implement FigureCanvasBackend (the thing that controls the figure and handles the events if any) and FigureManagerBackend (wraps the canvas and the window for MATLAB interface). FigureCanvasBase implements a backend switching mechanism

* Figure is now an Artist (like everything else in the figure) and is totally backend independent

* GDFONTPATH renamed to TTFPATH

* backend faceColor argument changed to rgbFace

* colormap stuff moved to colors.py

* arg_to_rgb in backend_bases moved to class ColorConverter in colors.py

* GD users must upgrade to gd-2.0.22 and gdmodule-0.52 since new gd features (clipping, antialiased lines) are now used.

* Renderer must implement points_to_pixels

Migrating code:

MATLAB interface:

The only API change for those using the MATLAB interface is in how you call figure redraws for dynamically updating figures. In the old API, you did

```python
fig.draw()
```

In the new API, you do

```python
manager = get_current_fig_manager()
manager.canvas.draw()
```

See the examples system_monitor.py, dynamic_demo.py, and anim.py
API

There is one important API change for application developers. Figure instances used subclass GUI widgets that enabled them to be placed directly into figures. e.g., FigureGTK subclassed gtk.DrawingArea. Now the Figure class is independent of the backend, and FigureCanvas takes over the functionality formerly handled by Figure. In order to include figures into your apps, you now need to do, for example

```python
# gtk example
fig = Figure(figsize=(5,4), dpi=100)
canvas = FigureCanvasGTK(fig)  # a gtk.DrawingArea
canvas.show()
vbox.pack_start(canvas)
```

If you use the NavigationToolbar, this in now intialized with a FigureCanvas, not a Figure. The examples embedding_in_gtk.py, embedding_in_gtk2.py, and mpl_with_glade.py all reflect the new API so use these as a guide.

All prior calls to

```python
figure.draw() and figure.print_figure(args)
```

should now be

```python
canvas.draw() and canvas.print_figure(args)
```

Apologies for the inconvenience. This refactorization brings significant more freedom in developing matplotlib and should bring better plotting capabilities, so I hope the inconvenience is worth it.

34.35 Changes for 0.42

* Refactoring AxisText to be backend independent. Text drawing and get_window_extent functionality will be moved to the Renderer.

* backend_bases.AxisTextBase is now text.Text module

* All the erase and reset functionalty removed from AxisText - not needed with double buffered drawing. Ditto with state change. Text instances have a get_prop_tup method that returns a hashable tuple of text properties which you can use to see if text props have changed, e.g., by caching a font or layout instance in a dict with the prop tup as a key -- see RendererGTK.get_pango_layout in backend_gtk for an example.
* Text._get_xy_display renamed Text.get_xy_display

* Artist set_renderer and wash_brushes methods removed

* Moved Legend class from matplotlib.axes into matplotlib.legend

* Moved Tick, XTick, YTick, Axis, XAxis, YAxis from matplotlib.axes to matplotlib.axis

* moved process_text_args to matplotlib.text

* After getting Text handled in a backend independent fashion, the import process is much cleaner since there are no longer cyclic dependencies

* matplotlib.matlab._get_current_fig_manager renamed to matplotlib.matlab.get_current_fig_manager to allow user access to the GUI window attribute, e.g., figManager.window for GTK and figManager.frame for wx

### 34.36 Changes for 0.40

- Artist
  * __init__ takes a DPI instance and a Bound2D instance which is the bounding box of the artist in display coords
  * get_window_extent returns a Bound2D instance
  * set_size is removed; replaced by bbox and dpi
  * the clip_gc method is removed. Artists now clip themselves with their box
  * added _clipOn boolean attribute. If True, gc clip to bbox.

- AxisTextBase
  * Initialized with a transx, transy which are Transform instances
  * set_drawing_area removed
  * get_left_right and get_top_bottom are replaced by get_window_extent

- Line2D Patches now take transx, transy
  * Initialized with a transx, transy which are Transform instances

- Patches
  * Initialized with a transx, transy which are Transform instances

- FigureBase attributes dpi is a DPI intance rather than scalar and new attribute bbox is a Bound2D in display coords, and I got rid of the left, width, height, etc... attributes. These are now accessible as, for example, bbox.x.min is left, bbox.x.interval() is width, bbox.y.max is top, etc...

- GcfBase attribute pagesize renamed to figsize

- Axes
* removed figbg attribute
* added fig instance to __init__
* resizing is handled by figure call to resize.

- Subplot
  * added fig instance to __init__

- Renderer methods for patches now take gcEdge and gcFace instances.
  gcFace=None takes the place of filled=False

- True and False symbols provided by cbook in a python2.3 compatible way

- new module transforms supplies Bound1D, Bound2D and Transform instances and more

- Changes to the MATLAB helpers API

  * _matlab_helpers.GcfBase is renamed by Gcf. Backends no longer need to derive from this class. Instead, they provide a factory function new_figure_manager(num, figsize, dpi). The destroy method of the GcfDerived from the backends is moved to the derived FigureManager.

  * FigureManagerBase moved to backend_bases

  * Gcf.get_all_figwins renamed to Gcf.get_all_fig_managers

Jeremy:

Make sure to self._reset = False in AxisTextWX._set_font. This was something missing in my backend code.
**THE TOP LEVEL MATPLOTLIB MODULE**

```python
matplotlib.use(arg, warn=True, force=False)
```

Set the matplotlib backend to one of the known backends.

The argument is case-insensitive. `warn` specifies whether a warning should be issued if a backend has already been set up. `force` is an experimental flag that tells matplotlib to attempt to initialize a new backend by reloading the backend module.

**Note:** This function must be called before importing pyplot for the first time; or, if you are not using pyplot, it must be called before importing matplotlib.backends. If `warn` is True, a warning is issued if you try and call this after pylab or pyplot have been loaded. In certain black magic use cases, e.g. `pyplot.switch_backend()`, we are doing the reloading necessary to make the backend switch work (in some cases, e.g., pure image backends) so one can set `warn=False` to suppress the warnings.

To find out which backend is currently set, see `matplotlib.get_backend()`.

```python
matplotlib.get_backend()
```

Return the name of the current backend.

```python
matplotlib.rcParams
```

An instance of `RcParams` for handling default matplotlib values.

```python
matplotlib.rc(group, **kwargs)
```

Set the current rc params. Group is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., `(xtick, ytick)`. `kwargs` is a dictionary attribute name/value pairs, e.g.:

```python
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```python
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:
<table>
<thead>
<tr>
<th>Alias</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>'lw'</td>
<td>'linewidth'</td>
</tr>
<tr>
<td>'ls'</td>
<td>'linestyle'</td>
</tr>
<tr>
<td>'c'</td>
<td>'color'</td>
</tr>
<tr>
<td>'fc'</td>
<td>'facecolor'</td>
</tr>
<tr>
<td>'ec'</td>
<td>'edgecolor'</td>
</tr>
<tr>
<td>'mew'</td>
<td>'markeredgewidth'</td>
</tr>
<tr>
<td>'aa'</td>
<td>'antialiased'</td>
</tr>
</tbody>
</table>

Thus you could abbreviate the above rc command as:

```python
r('lines', lw=2, c='r')
```

Note you can use python’s kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```python
font = {'family': 'monospace',
        'weight': 'bold',
        'size': 'larger'}
rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `rcdefaults()` to restore the default rc params after changes.

```
matplotlib.matplotlib_fname()
```

Get the location of the config file.

The file location is determined in the following order

- `$PWD/matplotlibrc`
- environment variable `MATPLOTLIBRC`
- `$MPLCONFIGDIR/matplotlib`
- On Linux,
  - `~/.matplotlib/matplotlibrc`, if it exists
  - or `$XDG_CONFIG_HOME/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is defined)
  - or `~/.config/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is not defined)
- On other platforms,
  - `~/.matplotlib/matplotlibrc` if `HOME` is defined.
- Lastly, it looks in `~/.matplotlibrc` for a system-defined copy.

```
class matplotlib.RcParams(*args, **kwargs)
```

A dictionary object including validation

validating functions are defined and associated with rc parameters in `matplotlib.rcsetup`

```
matplotlib.rc_params(fail_on_error=False)
```

Return a `matplotlib.RcParams` instance from the default matplotlib rc file.

```
matplotlib.rc_params_from_file(fname, fail_on_error=False, use_default_template=True)
```

Return `matplotlib.RcParams` from the contents of the given file.

**Parameters**
- `fname` : str
  Name of file parsed for matplotlib settings.
fail_on_error : bool
    If True, raise an error when the parser fails to convert a parameter.
use_default_template : bool
    If True, initialize with default parameters before updating with those in the given file. If False, the configuration class only contains the parameters specified in the file. (Useful for updating dicts.)

class matplotlib.rc_context(rc=None, fname=None)
    Return a context manager for managing rc settings.

This allows one to do:

```python
with mpl.rc_context(fname='screen.rc'):
    plt.plot(x, a)
    with mpl.rc_context(fname='print.rc'):
        plt.plot(x, b)
    plt.plot(x, c)
```

The ‘a’ vs ‘x’ and ‘c’ vs ‘x’ plots would have settings from ‘screen.rc’, while the ‘b’ vs ‘x’ plot would have settings from ‘print.rc’.

A dictionary can also be passed to the context manager:

```python
with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
    plt.plot(x, a)
```

The ‘rc’ dictionary takes precedence over the settings loaded from ‘fname’. Passing a dictionary only is also valid.
AFM (ADOBE FONT METRICS INTERFACE)

36.1 matplotlib.afm

This is a python interface to Adobe Font Metrics Files. Although a number of other python implementations exist, and may be more complete than this, it was decided not to go with them because they were either:

1. copyrighted or used a non-BSD compatible license
2. had too many dependencies and a free standing lib was needed
3. Did more than needed and it was easier to write afresh rather than figure out how to get just what was needed.

It is pretty easy to use, and requires only built-in python libs:

```python
>>> from matplotlib import rcParams
>>> import os.path

>>> afm_fname = os.path.join(rcParams['datapath'], ...
    'fonts', 'afm', 'ptmr8a.afm')

>>> from matplotlib.afm import AFM

>>> afm = AFM(open(afm_fname))

>>> afm.string_width_height('What the heck?')
(6220.0, 694)

>>> afm.get_fontname()
'Times-Roman'

>>> afm.get_kern_dist('A', 'f')
0

>>> afm.get_kern_dist('A', 'y')
-92.0

>>> afm.get_bbox_char('!')
[130, -9, 238, 676]
```

class matplotlib.afm.AFM(fh)

Bases: object

Parse the AFM file in file object \( fh \)

get_angle()

    Return the fontangle as float
get_bbox_char\((c, isord=False)\)

get_capheight()
  Return the cap height as float

get_familyname()
  Return the font family name, e.g., ‘Times’

get_fontname()
  Return the font name, e.g., ‘Times-Roman’

get_fullname()
  Return the font full name, e.g., ‘Times-Roman’

get_height_char\((c, isord=False)\)
  Get the height of character \(c\) from the bounding box. This is the ink height (space is 0)

get_horizontal_stem_width()
  Return the standard horizontal stem width as float, or \(None\) if not specified in AFM file.

get_kern_dist\((c1, c2)\)
  Return the kerning pair distance (possibly 0) for chars \(c1\) and \(c2\)

get_kern_dist_from_name\((name1, name2)\)
  Return the kerning pair distance (possibly 0) for chars \(name1\) and \(name2\)

get_name_char\((c, isord=False)\)
  Get the name of the character, i.e., ‘;’ is ‘semicolon’

get_str_bbox\((s)\)
  Return the string bounding box

get_str_bbox_and_descent\((s)\)
  Return the string bounding box

get_underline_thickness()
  Return the underline thickness as float

get_vertical_stem_width()
  Return the standard vertical stem width as float, or \(None\) if not specified in AFM file.

get_weight()
  Return the font weight, e.g., ‘Bold’ or ‘Roman’

get_width_char\((c, isord=False)\)
  Get the width of the character from the character metric WX field

get_width_from_char_name\((name)\)
  Get the width of the character from a type1 character name

get_xheight()
  Return the xheight as float

string_width_height\((s)\)
  Return the string width (including kerning) and string height as a \((w, h)\) tuple.
matplotlib.afm.parse_afm(fh)
Parse the Adobe Font Metics file in file handle fh. Return value is a (dhead, dcmetrics, dkernpairs, dcomposite) tuple where dhead is a _parse_header() dict, dcmetrics is a _parse_composites() dict, dkernpairs is a _parse_kern_pairs() dict (possibly {}), and dcomposite is a _parse_composites() dict (possibly {})
37.1 matplotlib.animation

class matplotlib.animation.AVConvBase
    Bases: matplotlib.animation.FFmpegBase
    args_key = u'animation.avconv_args'
    exec_key = u'animation.avconv_path'

class matplotlib.animation.AVConvFileWriter(*args, **kwargs)
    Bases: matplotlib.animation.AVConvBase, matplotlib.animation.FFmpegFileWriter

class matplotlib.animation.AVConvWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
    Bases: matplotlib.animation.AVConvBase, matplotlib.animation.FFmpegWriter

Construct a new MovieWriter object.

dfps: int Frame rate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

metadata: dict of string: string or None A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

class matplotlib.animation.Animation(fig, event_source=None, blit=False)
    Bases: object

    This class wraps the creation of an animation using matplotlib. It is only a base class which should be subclassed to provide needed behavior.
fig is the figure object that is used to get draw, resize, and any other needed events.

event_source is a class that can run a callback when desired events are generated, as well as be stopped and started. Examples include timers (see TimedAnimation) and file system notifications.

blit is a boolean that controls whether blitting is used to optimize drawing.

**new_frame_seq()**

Creates a new sequence of frame information.

**new_saved_frame_seq()**

Creates a new sequence of saved/cached frame information.

**save(filename, writer=None, fps=None, dpi=None, codec=None, bitrate=None, extra_args=None, metadata=None, extra_anim=None, savefig_kwars=None)**

Saves a movie file by drawing every frame.

*filename* is the output filename, e.g., `mymovie.mp4`

*writer* is either an instance of MovieWriter or a string key that identifies a class to use, such as ‘ffmpeg’ or ‘mencoder’. If nothing is passed, the value of the rcparam animation.writer is used.

*fps* is the frames per second in the movie. Defaults to None, which will use the animation’s specified interval to set the frames per second.

*dpi* controls the dots per inch for the movie frames. This combined with the figure’s size in inches controls the size of the movie.

*codec* is the video codec to be used. Not all codecs are supported by a given MovieWriter. If none is given, this defaults to the value specified by the rcparam animation.codec.

*bitrate* specifies the amount of bits used per second in the compressed movie, in kilobits per second. A higher number means a higher quality movie, but at the cost of increased file size. If no value is given, this defaults to the value given by the rcparam animation.bitrate.

*extra_args* is a list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

*metadata* is a dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

*extra_anim* is a list of additional Animation objects that should be included in the saved movie file. These need to be from the same matplotlib.Figure instance. Also, animation frames will just be simply combined, so there should be a 1:1 correspondence between the frames from the different animations.

*savefig_kwars* is a dictionary containing keyword arguments to be passed on to the ‘savefig’ command which is called repeatedly to save the individual frames. This can be used to set tight bounding boxes, for example.

**class matplotlib.animation.ArtistAnimation(fig, artists, *args, **kwargs)**

Bases: matplotlib.animation.TimedAnimation

Before calling this function, all plotting should have taken place and the relevant artists saved.

frame_info is a list, with each list entry a collection of artists that represent what needs to be enabled on each frame. These will be disabled for other frames.
class \texttt{matplotlib.animation.FFMpegBase}

\begin{verbatim}
args_key = u'animation.ffmpeg_args'  
exec_key = u'animation.ffmpeg_path'
output_args
\end{verbatim}

class \texttt{matplotlib.animation.FFMpegFileWriter}(*\texttt{args}, **\texttt{kwargs})  
Bases: \texttt{matplotlib.animation.FileMovieWriter, matplotlib.animation.FFMpegBase}

\begin{verbatim}
supported_formats = [u'png', u'jpeg', u'ppm', u'tiff', u'sgi', u'bmp', u'pbm', u'raw', u'rgba']
\end{verbatim}

class \texttt{matplotlib.animation.FFMpegWriter}(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)  
Bases: \texttt{matplotlib.animation.MovieWriter, matplotlib.animation.FFMpegBase}

Construct a new MovieWriter object.

\textbf{fps: int} Frame rate for movie.

\textbf{codec: string or None, optional} The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

\textbf{bitrate: int or None, optional} The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

\textbf{extra_args: list of strings or None} A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

\textbf{metadata: dict of string:string or None} A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

class \texttt{matplotlib.animation.FileMovieWriter}(*\texttt{args}, **\texttt{kwargs})  
Bases: \texttt{matplotlib.animation.MovieWriter}

\texttt{MovieWriter} subclass that handles writing to a file.

\texttt{cleanup()}

\texttt{finish()}

\texttt{frame_format}

Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

\texttt{grab_frame(**savefig_kwargs)}

Grab the image information from the figure and save as a movie frame. All keyword arguments in
savefig_kwargs are passed on to the ‘savefig’ command that saves the figure.

```python
setup(fig, outfile, dpi, frame_prefix=u'_tmp', clear_temp=True)
```

Perform setup for writing the movie file.

- **fig**: matplotlib.Figure instance The figure object that contains the information for frames
- **outfile**: string The filename of the resulting movie file
- **dpi**: int The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.
- **frame_prefix**: string, optional The filename prefix to use for the temporary files. Defaults to ‘_tmp’
- **clear_temp**: bool Specifies whether the temporary files should be deleted after the movie is written. (Useful for debugging.) Defaults to True.

```python
class matplotlib.animation.FuncAnimation(fig, func, frames=None, init_func=None, fargs=None, save_count=None, **kwargs)
```

Makes an animation by repeatedly calling a function `func`, passing in (optional) arguments in `fargs`.

- **frames** can be a generator, an iterable, or a number of frames.
- **init_func** is a function used to draw a clear frame. If not given, the results of drawing from the first item in the frames sequence will be used. This function will be called once before the first frame.
- If `blit=True`, `func` and `init_func` should return an iterable of drawables to clear.

- **kwargs** include `repeat`, `repeat_delay`, and `interval`:
  - `interval` draws a new frame every `interval` milliseconds.
  - `repeat` controls whether the animation should repeat when the sequence of frames is completed.
  - `repeat_delay` optionally adds a delay in milliseconds before repeating the animation.

```python
new_frame_seq()
new_saved_frame_seq()
```

class matplotlib.animation.ImageMagickBase

```python
class matplotlib.animation.ImageMagickFileWriter(*args, **kwargs)
```

Bases: matplotlib.animation.FileMovieWriter, matplotlib.animation.ImageMagickBase
supported_formats = [u'png', u'jpeg', u'ppm', u'tiff', u'sgi', u'bmp', u'pbm', u'raw', u'rgba']

class matplotlib.animation.ImageMagickWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
Bases: matplotlib.animation.MovieWriter, matplotlib.animation.ImageMagickBase

Construct a new MovieWriter object.

fps: int Frame rate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

metadata: dict of string: string or None A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

class matplotlib.animation.MencoderBase

allowed_metadata = [u'name', u'artist', u'genre', u'subject', u'copyright', u'srcform', u'comment']

args_key = u'animation.mencoder_args'

exec_key = u'animation.mencoder_path'

output_args

class matplotlib.animation.MencoderFileWriter(*args, **kwargs)
Bases: matplotlib.animation.FileMovieWriter, matplotlib.animation.MencoderBase

supported_formats = [u'png', u'jpeg', u'tga', u'sgi']

class matplotlib.animation.MencoderWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
Bases: matplotlib.animation.MovieWriter, matplotlib.animation.MencoderBase

Construct a new MovieWriter object.

fps: int Frame rate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam
animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args:** *list of strings or None* A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

**metadata:** *dict of string:string or None* A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

```python
class matplotlib.animation.MovieWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

Bases: object

Base class for writing movies. Fundamentally, what a MovieWriter does is provide a way to grab frames by calling grab_frame(). setup() is called to start the process and finish() is called afterwards. This class is set up to provide for writing movie frame data to a pipe. saving() is provided as a context manager to facilitate this process as:

```python
with moviewriter.saving('myfile.mp4 '):
    # Iterate over frames
    moviewriter.grab_frame()
```

The use of the context manager ensures that setup and cleanup are performed as necessary.

**frame_format:** *string* The format used in writing frame data, defaults to ‘rgba’

**fps:** *int* Framerate for movie.

**codec:** *string or None, optional* The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

**bitrate:** *int or None, optional* The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args:** *list of strings or None* A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

**metadata:** *dict of string:string or None* A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**classmethod bin_path()**

Returns the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

**cleanup()**

Clean-up and collect the process used to write the movie file.

**finish()**

Finish any processing for writing the movie.

**frame_size**

A tuple (width, height) in pixels of a movie frame.
grab_frame(**savefig_kwargs)

Grab the image information from the figure and save as a movie frame. All keyword arguments in
savefig_kwargs are passed on to the `savefig` command that saves the figure.

classmethod isAvailable()

Check to see if a MovieWriter subclass is actually available by running the commandline tool.

saving(*args, **kwds)

Context manager to facilitate writing the movie file.

*args are any parameters that should be passed to setup.

setup(fig, outfile, dpi, *args)

Perform setup for writing the movie file.

fig: `matplotlib.Figure` instance The figure object that contains the information for frames
outfile: string The filename of the resulting movie file
dpi: int The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie
file.

class matplotlib.animation.MovieWriterRegistry

Bases: object

is_available(name)

list()

Get a list of available MovieWriters.

register(name)

class matplotlib.animation.TimedAnimation(fig, interval=200, repeat_delay=0, repeat=True, event_source=None, *args, **kwargs)

Bases: matplotlib.animation.Animation

Animation subclass that supports time-based animation, drawing a new frame every `interval` milliseconds.

repeat controls whether the animation should repeat when the sequence of frames is completed.

repeat_delay optionally adds a delay in milliseconds before repeating the animation.
class matplotlib.artist.Artist
    Bases: object

    Abstract base class for someone who renders into a FigureCanvas.

    add_callback(func)
        Adds a callback function that will be called whenever one of the Artist's properties changes.
        Returns an id that is useful for removing the callback with remove_callback() later.

    contains(mouseevent)
        Test whether the artist contains the mouse event.
        Returns the truth value and a dictionary of artist specific details of selection, such as which points are contained in the pick radius. See individual artists for details.

    convert_xunits(x)
        For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

    convert_yunits(y)
        For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

    draw(renderer, *args, **kwargs)
        Derived classes drawing method

    findobj(match=None, include_self=True)
        Find artist objects.
        Recursively find all Artist instances contained in self.
        match can be
        • None: return all objects contained in artist.
        • function with signature boolean = match(artist) used to filter matches
        • class instance: e.g., Line2D. Only return artists of class type.
        If include_self is True (default), include self in the list to be checked for a match.

    get_agg_filter()
        return filter function to be used for agg filter

    get_alpha()
        Return the alpha value used for blending - not supported on all backends

    get_animated()
        Return the artist’s animated state

    get_axes()
        Return the Axes instance the artist resides in, or None

    get_children()
        Return a list of the child Artist's this :class:`Artist contains.
get_clip_box()  
Returns artist clipbox

get_clip_on()  
Returns whether artist uses clipping

get_clip_path()  
Returns artist clip path

get_contains()  
Returns the _contains test used by the artist, or None for default.

get_figure()  
Returns the Figure instance the artist belongs to.

get_gid()  
Returns the group id

get_label()  
Get the label used for this artist in the legend.

get_path_effects()  

get_picker()  
Return the picker object used by this artist

get_rasterized()  
return True if the artist is to be rasterized

get_sketch_params()  
Returns the sketch parameters for the artist.

    Returns sketch_params : tuple or None

    A 3-tuple with the following elements:
    • scale: The amplitude of the wiggle perpendicular to the source line.
    • length: The length of the wiggle along the line.
    • randomness: The scale factor by which the length is shrunken or expanded.

    May return None if no sketch parameters were set.

get_snap()  
Returns the snap setting which may be:

    • True: snap vertices to the nearest pixel center
    • False: leave vertices as-is
    • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

    Only supported by the Agg and MacOSX backends.

get_transform()  
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()  
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.
get_url()
Returns the url

get_visible()
Return the artist’s visibility

get_window_extent(renderer)
Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

get_zorder()
Return the Artist's zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
call signature:

    pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
Return True if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.
remove_callback(oid)
Remove a callback based on its id.

See also:

add_callback() For adding callbacks

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha value used for blending - not supported on all backends.

ACCEPTS: float (0.0 transparent through 1.0 opaque)

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_axes(axes)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

set_clip_box(clipbox)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
• a Patch (or subclass) instance
  • a Path instance, in which case an optional Transform instance may be provided, which will
    be applied to the path before using it for clipping.
  • None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping
box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [ (Path, Transform) | Patch | None ]

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which
determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)
If the mouse event is over the artist, return \( \text{hit} = \text{True} \) and \( \text{props} \) is a dictionary of properties you want returned with the contains test.

**ACCEPTS:** a callable function

**set_figure**(*fig*)

Set the Figure instance the artist belongs to.

**ACCEPTS:** a matplotlib.figure.Figure instance

**set_gid**(*gid*)

Sets the (group) id for the artist

**ACCEPTS:** an id string

**set_label**(*s*)

Set the label to \( s \) for auto legend.

**ACCEPTS:** string or anything printable with ‘\%s’ conversion.

**set_lod**(*on*)

Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

**ACCEPTS:** \[True \mid False\]

**set_path_effects**(*path_effects*)

Set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

**set_picker**(*picker*)

Set the epsilon for picking used by this artist

**picker** can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if \( \text{True} \) then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  \[
  \text{hit, props} = \text{picker(} \text{artist, mouseevent} \text{)}
  \]

  to determine the hit test. if the mouse event is over the artist, return \( \text{hit} = \text{True} \) and \( \text{props} \) is a dictionary of properties you want added to the PickEvent attributes.

**ACCEPTS:** \[\text{None} \mid \text{float} \mid \text{boolean} \mid \text{callable}\]

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior
ACCEPTS: [True | False | None]

**set_sketch_params**(scale=None, length=None, randomness=None)
Sets the sketch parameters.

Parameters:
- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.
- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)
- **randomness**: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
  Only supported by the Agg and MacOSX backends.

**set_transform**(t)
Set the Transform instance used by this artist.

Accepts: Transform instance

**set_url**(url)
Sets the url for the artist

Accepts: a url string

**set_visible**(b)
Set the artist’s visibility.

Accepts: [True | False]

**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Accepts: any number

**update**(props)
Update the properties of this Artist from the dictionary prop.

**update_from**(other)
Copy properties from other to self.

**zorder** = 0

**class** matplotlib.artist.ArtistInspector(o)
A helper class to inspect an Artist and return information about it’s settable properties and their current values.

Initialize the artist inspector with an Artist or sequence of Artists. If a sequence is used, we assume it is a homogeneous sequence (all Artists are of the same type) and it is your responsibility to make sure this is so.
aliased_name(s)
return ‘PROPNAME or alias’ if s has an alias, else return PROPNAME.
e.g., for the line markerfacecolor property, which has an alias, return ‘markerfacecolor or mfc’ and for the transform property, which does not, return ‘transform’

aliased_name_rest(s, target)
return ‘PROPNAME or alias’ if s has an alias, else return PROPNAME formatted for ReST
e.g., for the line markerfacecolor property, which has an alias, return ‘markerfacecolor or mfc’ and for the transform property, which does not, return ‘transform’

findobj(match=None)
Recursively find all matplotlib.artist.Artist instances contained in self.
If match is not None, it can be
• function with signature boolean = match(artist)
• class instance: e.g., Line2D
used to filter matches.

get_aliases()
Get a dict mapping fullname -> alias for each alias in the ArtistInspector.
e.g., for lines:

{'markerfacecolor': 'mfc',
 'linewidth' : 'lw',
}

get_setters()
Get the attribute strings with setters for object. e.g., for a line, return ['markerfacecolor',
 'linewidth', ....].

get_valid_values(attr)
Get the legal arguments for the setter associated with attr.
This is done by querying the docstring of the function set_attr for a line that begins with ACCEPTS:
e.g., for a line linestyle, return “[ '‐' | '‐‐' | '‐‐‐' | ':' | 'steps' | 'None' ]”

is_alias(o)
Return True if method object o is an alias for another function.

pprint_getters()
Return the getters and actual values as list of strings.

pprint_setters(prop=None, leadingspace=2)
If prop is None, return a list of strings of all settable properies and their valid values.
If prop is not None, it is a valid property name and that property will be returned as a string of property : valid values.

pprint_setters_rest(prop=None, leadingspace=2)
If prop is None, return a list of strings of all settable properies and their valid values. Format the output for ReST
If prop is not None, it is a valid property name and that property will be returned as a string of property : valid values.

```
properties()
return a dictionary mapping property name -> value
```

```
matplotlib.artist.allow_rasterization(draw)
Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependant renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.
```

```
matplotlib.artist.get(obj, property=None)
Return the value of object’s property. property is an optional string for the property you want to return
Example usage:

```
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

obj is a Artist instance, e.g., Line2D or an instance of a Axes or matplotlib.text.Text. If the property is ‘somename’, this function returns

```
obj.get_somename()
```

getp() can be used to query all the gettable properties with getp(obj). Many properties have aliases for shorter typing, e.g. ‘lw’ is an alias for ‘linewidth’. In the output, aliases and full property names will be listed as:

```
property or alias = value
```

e.g.:

```
linewidth or lw = 2
```

```
matplotlib.artist.getp(obj, property=None)
Return the value of object’s property. property is an optional string for the property you want to return
Example usage:

```
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

obj is a Artist instance, e.g., Line2D or an instance of a Axes or matplotlib.text.Text. If the property is ‘somename’, this function returns

```
obj.get_somename()
```

getp() can be used to query all the gettable properties with getp(obj). Many properties have aliases for shorter typing, e.g. ‘lw’ is an alias for ‘linewidth’. In the output, aliases and full property names will be listed as:

```
property or alias = value
```

e.g.:

```
linewidth or lw = 2
```

```
matplotlib.artist.kwdoc(a)
```

```
matplotlib.artist.setp(obj, *args, **kwargs)
Set a property on an artist object.
```
matplotlib supports the use of `setp()` (“set property”) and `getp()` to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```python
>>> line, = plot([1,2,3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```python
>>> setp(line, 'linestyle')
linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
```

If you want to see all the properties that can be set, and their possible values, you can do:

```python
>>> setp(line)
... long output listing omitted
```

`setp()` operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. E.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```python
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

`setp()` works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```python
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')       # python style
```
39.1 matplotlib.axes

```python
class matplotlib.axes.Axes(fig, rect, axisbg=None, frameon=True, sharex=None, sharey=None, label='', xscale=None, yscale=None, **kwargs)
```

The `Axes` contains most of the figure elements: `Axis`, `Tick`, `Line2D`, `Text`, `Polygon`, etc., and sets the coordinate system.

The `Axes` instance supports callbacks through a `callbacks` attribute which is a `CallbackRegistry` instance. The events you can connect to are ‘xlim_changed’ and ‘ylim_changed’ and the callback will be called with `func(ax)` where `ax` is the `Axes` instance.

```python
acorr(x, **kwargs)
```

Plot the autocorrelation of `x`.

**Parameters**

- `x` : sequence of scalar
  - `hold` : boolean, optional, default: True
  - `detrend` : callable, optional, default: `mlab.detrend_none`
    - `x` is detrended by the `detrend` callable. Default is no normalization.
  - `normed` : boolean, optional, default: True
    - if True, normalize the data by the autocorrelation at the 0-th lag.
  - `usevlines` : boolean, optional, default: True
    - if True, `Axes.vlines` is used to plot the vertical lines from the origin to the `acorr`. Otherwise, `Axes.plot` is used.
  - `maxlags` : integer, optional, default: 10
    - number of lags to show. If None, will return all `2 * len(x) - 1` lags.

**Returns**

- `(lags, c, line, b)` : where:
  - `lags` are a length `2*maxlags+1` lag vector.
  - `c` is the `2*maxlags+1` autocorrelation vector
  - `line` is a `Line2D` instance returned by `plot`.
  - `b` is the x-axis.

**Other Parameters**

- `linestyle` : `Line2D` prop, optional, default: None
  - Only used if `usevlines` is False.
- `marker` : string, optional, default: ‘o’
Notes

The cross correlation is performed with `numpy.correlate()` with `mode = 2`.

Examples

`xcorr` is top graph, and `acorr` is bottom graph.

```
add_artist(a)
Add any Artist to the axes.

Use add_artist only for artists for which there is no dedicated “add” method; and if necessary, use a method such as `update_datalim` or `update_datalim_numerix` to manually update the dataLim if the artist is to be included in autoscaling.

Returns the artist.

add_callback(func)
Adds a callback function that will be called whenever one of the Artist’s properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

add_collection(collection, autolim=True)
Add a Collection instance to the axes.
Returns the collection.

**add_container**(container)
Add a `Container` instance to the axes.
Returns the collection.

**add_image**(image)
Add a `AxesImage` to the axes.
Returns the image.

**add_line**(line)
Add a `Line2D` to the list of plot lines
Returns the line.

**add_patch**(p)
Add a `Patch` $p$ to the list of axes patches; the clipbox will be set to the Axes clipping box. If the transform is not set, it will be set to `transData`.
Returns the patch.

**add_table**(tab)
Add a `Table` instance to the list of axes tables
Returns the table.

aname = u’Artist’

**angle_spectrum**(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, **kwargs)
Plot the angle spectrum.
Call signature:

```python
angle_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
               pad_to=None, sides='default', **kwargs)
```

Compute the angle spectrum (wrapped phase spectrum) of $x$. Data is padded to a length of `$pad_to$` and the windowing function `$window$` is applied to the signal.

- $x$: 1-D array or sequence Array or sequence containing the data

Keyword arguments:

- **Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
- **window**: callable or ndarray A function or a vector of length `$NFFT$`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
sides: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

pad_to: integer
The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

Fc: integer The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple \( (spectrum, freqs, line) \):

spectrum: 1-D array The values for the angle spectrum in radians (real valued)
freqs: 1-D array The frequencies corresponding to the elements in spectrum
line: a Line2D instance The line created by this function

KWARGS control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘-‘</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

![Example plots](image-url)
See also:

* `magnitude_spectrum()`: plots the magnitudes of the corresponding frequencies.
* `phase_spectrum()`: plots the unwrapped version of this function.
* `specgram()`: can plot the angle spectrum of segments within the signal in a colormap.

**annotate(***args*, **kwargs*)

Create an annotation: a piece of text referring to a data point.

**Parameters**

s : string
    label

xy : (x, y)
    position of element to annotate

xytext : (x, y), optional, default: None
    position of the label s

xycoords : string, optional, default: “data”
    string that indicates what type of coordinates xy is. Examples: “figure points”, “figure pixels”, “figure fraction”, “axes points”, .... See `matplotlib.text.Annotation` for more details.

textcoords : string, optional
    string that indicates what type of coordinates text is. Examples: “figure points”, “figure pixels”, “figure fraction”, “axes points”, .... See `matplotlib.text.Annotation` for more details. Default is None.

arrowprops : `matplotlib.lines.Line2D` properties, optional
    Dictionary of line properties for the arrow that connects the annotation to the point. If the dictionary has a key `arrowstyle`, a `FancyArrowPatch` instance is created and drawn. See `matplotlib.text.Annotation` for more details on valid options. Default is None.

**Returns**

a : `Annotation`

**Notes**

`arrowprops`, if not `None`, is a dictionary of line properties (see `matplotlib.lines.Line2D`) for the arrow that connects annotation to the point.

If the dictionary has a key `arrowstyle`, a `FancyArrowPatch` instance is created with the given dictionary and is drawn. Otherwise, a `YAArrow` patch instance is created and drawn. Valid keys for `YAArrow` are:
Matplotlib, Release 1.4.3

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>head-width</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>oftentimes it is convenient to have the arrowtip and base a bit away from the text and point being annotated. If $d$ is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shrink percent of the distance $d$ away from the endpoints. i.e., $\text{shrink}=0.05$ is 5%</td>
</tr>
<tr>
<td>?</td>
<td>any key for matplotlib.patches.polygon</td>
</tr>
</tbody>
</table>

Valid keys for FancyArrowPatch are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
<tr>
<td>?</td>
<td>any key for matplotlib.patches.PathPatch</td>
</tr>
</tbody>
</table>

`xycoords` and `textcoords` are strings that indicate the coordinates of `xy` and `xtex`, and may be one of the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>‘offset points’</td>
<td>Specify an offset (in points) from the xy value</td>
</tr>
<tr>
<td>‘polar’</td>
<td>you can specify $\theta$, $r$ for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native “data” coordinate system.</td>
</tr>
</tbody>
</table>
If a 'points' or 'pixels' option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. e.g.:

```python
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of `Transform` or `Artist`. See `Annotating Axes` for more details.

The `annotation_clip` attribute controls the visibility of the annotation when it goes outside the axes area. If `True`, the annotation will only be drawn when the `xy` is inside the axes. If `False`, the annotation will always be drawn regardless of its position. The default is `None`, which behave as `True` only if `xycoords` is “data”.

Additional kwargs are `Text` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>backgroundcolor</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>bbox</code></td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>family</code></td>
<td>[FONTNAME</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fontproperties</code></td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>horizontalalignment</code></td>
<td>[‘center’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linespacing</code></td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>multialignment</code></td>
<td>[‘left’</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>position</code></td>
<td>(x,y)</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>rotation</code></td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td><code>rotation_mode</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>size</code> or <code>fontsize</code></td>
<td>[size in points</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>stretch</code> or <code>fontstretch</code></td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td><code>style</code> or <code>fontstyle</code></td>
<td>[‘normal’</td>
</tr>
<tr>
<td><code>text</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Examples**

![Graph with annotations](image_url)
apply_aspect(position=None)
Use _aspect() and _adjustable() to modify the axes box or the view limits.

arrow(x, y, dx, dy, **kwargs)
Add an arrow to the axes.

Call signature:

arrow(x, y, dx, dy, **kwargs)

Draws arrow on specified axis from (x, y) to (x + dx, y + dy). Uses FancyArrow patch to construct the arrow.

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use annotate() for example:

ax.annotate('', xy=(0.5, 0.5), xytext=(0, 0),
arrowprops=dict(arrowstyle='->'))

Optional kwargs control the arrow construction and properties:

Constructor arguments

width: float (default: 0.001) width of full arrow tail

length_includes_head: [True | False] (default: False) True if head is to be counted in calculating the length.
head_width: float or None (default: 3*width) total width of the full arrow head
head_length: float or None (default: 1.5 * head_width) length of arrow head
shape: ['full', 'left', 'right'] (default: 'full') draw the left-half, right-half, or full arrow
overhang: float (default: 0) fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.
head_starts_at_zero: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from Patch) are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
autoscale(enable=True, axis='both', tight=None)
Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.

**enable:** [True | False | None] True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis:** ['x' | 'y' | 'both'] which axis to operate on; default is ‘both’

**tight:** [True | False | None] If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat **tight** as False. The **tight** setting is retained for future autoscaling until it is explicitly changed.

Returns None.

autoscale_view(tight=None, scalex=True, scaley=True)
Autoscale the view limits using the data limits. You can selectively autoscale only a single axis, e.g., the xaxis by setting **scaley** to False. The autoscaling preserves any axis direction reversal that has already been done.

The data limits are not updated automatically when artist data are changed after the artist has been added to an Axes instance. In that case, use matplotlib.axes.Axes.relim() prior to calling autoscale_view.

axhline(y=0, xmin=0, xmax=1, **kwargs)
Add a horizontal line across the axis.

**Parameters**

**y** : scalar, optional, default: 0
y position in data coordinates of the horizontal line.

**xmin** : scalar, optional, default: 0
Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**xmax** : scalar, optional, default: 1
Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**Returns** Line2D

**See also:**

`axhspan` for example plot and source code

**Notes**

kwargs are the same as kwargs to `plot`, and can be used to control the line properties. e.g.,

**Examples**

• draw a thick red hline at ‘y’ = 0 that spans the xrange:

```python
>>> axhline(linewidth=4, color='r')
```

• draw a default hline at ‘y’ = 1 that spans the xrange:

```python
>>> axhline(y=1)
```

• draw a default hline at ‘y’ = .5 that spans the the middle half of the xrange:

```python
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Valid kwargs are Line2D properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
</tbody>
</table>

Continued on next page
Matplotlib, Release 1.4.3

Table 39.4 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgcolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)**

Add a horizontal span (rectangle) across the axis.

Call signature:

axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)

y coords are in data units and x coords are in axes (relative 0-1) units.

Draw a horizontal span (rectangle) from ymin to ymax. With the default values of xmin = 0 and xmax = 1, this always spans the xrange, regardless of the xlim settings, even if you change them, e.g., with the set_xlim() command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the y location is in data coordinates.

Return value is a matplotlib.patches.Polygon instance.

Examples:

- draw a gray rectangle from y = 0.25-0.75 that spans the horizontal extent of the axes:
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)

Valid kwargs are Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[’miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>[’solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
axis(*v, **kwargs)
Convenience method for manipulating the x and y view limits and the aspect ratio of the plot. For details, see axis().

kwargs are passed on to set_xlim() and set_ylim()

axvline(x=0, ymin=0, ymax=1, **kwargs)
Add a vertical line across the axes.

Parameters:
x : scalar, optional, default: 0
  x position in data coordinates of the vertical line.

ymin : scalar, optional, default: 0
  Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

ymax : scalar, optional, default: 1
  Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

Returns:
Line2D

See also:
axhspan for example plot and source code

Examples

- draw a thick red vline at x = 0 that spans the yrange:
>>> axvline(linewidth=4, color='r')

• draw a default vline at \(x = 1\) that spans the yrange:

>>> axvline(x=1)

• draw a default vline at \(x = 0.5\) that spans the middle half of the yrange:

>>> axvline(x=.5, ymin=0.25, ymax=0.75)

Valid kwargs are :class:`Line2D` properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an :class:<code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a :class:<code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a :class:<code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘-‘</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function (fn(artist, event))</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
</tbody>
</table>
Table 39.6 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)**

Add a vertical span (rectangle) across the axes.

Call signature:

```python
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

$x$ coords are in data units and $y$ coords are in axes (relative 0-1) units.

Draw a vertical span (rectangle) from $xmin$ to $xmax$. With the default values of $ymin = 0$ and $ymax = 1$, this always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the `set_ylim()` command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the y location is in data coordinates.

Return value is the `matplotlib.patches.Polygon` instance.

Examples:

- draw a vertical green translucent rectangle from $x=1.25$ to 1.55 that spans the yrange of the axes:

  ```python
  >>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
  ```

Valid kwargs are `Polygon` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
</tbody>
</table>
Table 39.7 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

axhspan() for example plot and source code

bar(left, height, width=0.8, bottom=None, **kwargs)

Make a bar plot.

Make a bar plot with rectangles bounded by:

left, left + width, bottom, bottom + height (left, right, bottom and top edges)

Parameters left : sequence of scalars
  the x coordinates of the left sides of the bars
height : sequence of scalars
  the heights of the bars
width : scalar or array-like, optional, default: 0.8
  the width(s) of the bars
bottom : scalar or array-like, optional, default: None
  the y coordinate(s) of the bars
color : scalar or array-like, optional
  the colors of the bar faces
edgecolor : scalar or array-like, optional
  the colors of the bar edges
linewidth : scalar or array-like, optional, default: None
  width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.
xerr : scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**yerr**: scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**ecolor**: scalar or array-like, optional, default: None
specifies the color of errorbar(s)

**capsize**: integer, optional, default: 3
determines the length in points of the error bar caps

**error_kw**:
dictionary of kwargs to be passed to errorbar method. *ecolor*
and *capsize* may be specified here rather than as independent
kwargs.

**align**: ['edge' | 'center'], optional, default: 'edge'
If edge, aligns bars by their left edges (for vertical bars) and
by their bottom edges (for horizontal bars). If center, interpret
the left argument as the coordinates of the centers of the bars.

**orientation**: ‘vertical’ | ‘horizontal’, optional, default: ‘vertical’
The orientation of the bars.

**log**: boolean, optional, default: False
If true, sets the axis to be log scale

**Returns**: *matplotlib.patches.Rectangle* instances.

**See also**: *barh* Plot a horizontal bar plot.

**Notes**

The optional arguments *color*, *edgecolor*, *linewidth*, *xerr*, and *yerr* can be either scalars
or sequences of length equal to the number of bars. This enables you to use bar as the ba-
sis for stacked bar charts, or candlestick plots. Detail: *xerr* and *yerr* are passed directly to
*errorbar()*, so they can also have shape 2xN for independent specification of lower and upper
errors.

Other optional kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>agg_filter</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>alpha</strong></td>
<td>float or None</td>
</tr>
<tr>
<td><strong>animated</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>antialiased</strong> or aa</td>
<td>[True</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td><strong>capstyle</strong></td>
<td>['butt'</td>
</tr>
<tr>
<td><strong>clip_box</strong></td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td><strong>clip_on</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>clip_path</strong></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>a callable function</td>
</tr>
<tr>
<td><strong>edgecolor or ec</strong></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.8 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Examples

Example: A stacked bar chart.
barbs(*args, **kw)

Plot a 2-D field of barbs.

Call signatures:

barb(U, V, **kw)
barb(U, V, C, **kw)
barb(X, Y, U, V, **kw)
barb(X, Y, U, V, C, **kw)

Arguments:

X, Y: The x and y coordinates of the barb locations (default is head of barb; see pivot kwarg)
U, V: Give the x and y components of the barb shaft
C: An optional array used to map colors to the barbs

All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if len(X) and len(Y) match the column and row dimensions of U, then X and Y will be expanded with numpy.meshgrid().

U, V, C may be masked arrays, but masked X, Y are not supported at present.

Keyword arguments:

length: Length of the barb in points; the other parts of the barb are scaled against this. Default is 9
**pivot**: [‘tip’ | ‘middle’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*. Default is ‘tip’

**barbcolor**: [color | color sequence] Specifies the color all parts of the barb except any flags. This parameter is analogous to the *edgcolor* parameter for polygons, which can be used instead. However this parameter will override *facecolor*.

**flagcolor**: [color | color sequence] Specifies the color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However this parameter will override *facecolor*. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

**sizes**: A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:
- ‘spacing’ - space between features (flags, full/half barbs)
- ‘height’ - height (distance from shaft to top) of a flag or full barb
- ‘width’ - width of a flag, twice the width of a full barb
- ‘emptybarb’ - radius of the circle used for low magnitudes

**fill_empty**: A flag on whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False

**rounding**: A flag to indicate whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple. Default is True

**barb_increments**: A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.
- ‘half’ - half barbs (Default is 5)
- ‘full’ - full barbs (Default is 10)
- ‘flag’ - flags (default is 50)

**flip_barb**: Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for each individual barb. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere.) Default is False

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as shown schematically below:
The largest increment is given by a triangle (or “flag”). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

Linewidths and edgecolors can be used to customize the barb. Additional PolyCollection keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facer color or facer colors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Example:
**barh**(*bottom*, *width*, *height*=0.8, *left*=None, **kwargs)

Make a horizontal bar plot.

Make a horizontal bar plot with rectangles bounded by:

   **left**, **left** + **width**, **bottom**, **bottom** + **height** (left, right, bottom and top edges)

**bottom**, **width**, **height**, and **left** can be either scalars or sequences

**Parameters**

- **bottom**: scalar or array-like
  the y coordinate(s) of the bars
- **width**: scalar or array-like
  the width(s) of the bars
- **height**: sequence of scalars, optional, default: 0.8
  the heights of the bars
- **left**: sequence of scalars
  the x coordinates of the left sides of the bars

**Returns**

matplotlib.patches.Rectangle instances.

**Other Parameters**

- **color**: scalar or array-like, optional
  the colors of the bars
- **edgecolor**: scalar or array-like, optional
  the colors of the bar edges
- **linewidth**: scalar or array-like, optional, default: None
  width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.
xerr : scalar or array-like, optional, default: None
    if not None, will be used to generate errorbar(s) on the bar chart
yerr : scalar or array-like, optional, default: None
    if not None, will be used to generate errorbar(s) on the bar chart
ecolor : scalar or array-like, optional, default: None
    specifies the color of errorbar(s)
capsize : integer, optional, default: 3
    determines the length in points of the error bar caps

error_kw :
    dictionary of kwargs to be passed to errorbar method. ecolor
    and capsize may be specified here rather than as independent
    kwargs.
align : ['edge' | 'center'], optional, default: 'edge'
    If edge, aligns bars by their left edges (for vertical bars) and
    by their bottom edges (for horizontal bars). If center, interpret
    the left argument as the coordinates of the centers of the bars.
orientation : 'vertical' | 'horizontal', optional, default: 'vertical'
    The orientation of the bars.
log : boolean, optional, default: False
    If true, sets the axis to be log scale

See also:
bar Plot a vertical bar plot.

Notes

The optional arguments color, edgecolor, linewidth, xerr, and yerr can be either scalars
or sequences of length equal to the number of bars. This enables you to use bar as the ba-
sis for stacked bar charts, or candlestick plots. Detail: xerr and yerr are passed directly to
errorbar(), so they can also have shape 2xN for independent specification of lower and upper
errors.

Other optional kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.10 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

`boxplot(x, notch=False, sym=None, vert=True, whis=1.5, positions=None, widths=None, patch_artist=False, bootstrap=None, usermedians=None, conf_intervals=None, meanline=False, showmeans=False, showcaps=True, showbox=True, showfliers=True, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whiskerprops=None, manage_xticks=True)`

Make a box and whisker plot.

Call signature:

```
boxplot(self, x, notch=False, sym='b+', vert=True, whis=1.5, positions=None, widths=None, patch_artist=False, bootstrap=None, usermedians=None, conf_intervals=None, meanline=False, showmeans=False, showcaps=True, showbox=True, showfliers=True, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whiskerprops=None, manage_xticks=True):
```

Make a box and whisker plot for each column of x or each vector in sequence x. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

**Parameters** x : Array or a sequence of vectors.

The input data.

notch [bool, default = False] If False, produces a rectangular box plot. If True, will produce a notched box plot.
sym [str or None, default = None] The default symbol for flier points. Enter an empty string ("") if you don’t want to show fliers. If None, then the fliers default to 'b+' If you want more control use the flierprops kwarg.

vert [bool, default = True] If True (default), makes the boxes vertical. If False, makes horizontal boxes.

whis [float, sequence (default = 1.5) or string] As a float, determines the reach of the whiskers past the first and third quartiles (e.g., Q3 + whis*IQR, IQR = interquartile range, Q3-Q1). Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, whis can be the string ‘range’ to force the whiskers to the min and max of the data. In the edge case that the 25th and 75th percentiles are equivalent, whis will be automatically set to ‘range’.

bootstrap [None (default) or integer] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap==None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine it’s 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians [array-like or None (default)] An array or sequence whose first dimension (or length) is compatible with x. This overrides the medians computed by matplotlib for each element of usermedians that is not None. When an element of usermedians == None, the median will be computed by matplotlib as normal.

conf_intervals [array-like or None (default)] Array or sequence whose first dimension (or length) is compatible with x and whose second dimension is 2. When the current element of conf_intervals is not None, the notch locations computed by matplotlib are overridden (assuming notch is True). When an element of conf_intervals is None, boxplot compute notches the method specified by the other kwargs (e.g., bootstrap).

positions [array-like, default = [1, 2, ..., n]] Sets the positions of the boxes. The ticks and limits are automatically set to match the positions.

widths [array-like, default = 0.5] Either a scalar or a vector and sets the width of each box. The default is 0.5,
or \(0.15 \times (\text{distance between extreme positions})\)
if that is smaller.

- **labels** [sequence or None (default)] Labels for each dataset. 
  Length must be compatible with dimensions of \(x\)
- **patch_artist** [bool, default = False] If False produces boxes  
  with the Line2D artist If True produces boxes with the  
  Patch artist
- **showmeans** [bool, default = False] If True, will toggle one the  
  rendering of the means
- **showcaps** [bool, default = True] If True, will toggle one the  
  rendering of the caps
- **showbox** [bool, default = True] If True, will toggle one the ren-  
  dering of box
- **showfliers** [bool, default = True] If True, will toggle one the  
  rendering of the fliers
- **boxprops** [dict or None (default)] If provided, will set the plot-  
  ting style of the boxes
- **whiskerprops** [dict or None (default)] If provided, will set the  
  plotting style of the whiskers
- **capprops** [dict or None (default)] If provided, will set the plot-  
  ting style of the caps
- **flierprops** [dict or None (default)] If provided, will set the plot-  
  ting style of the fliers
- **medianprops** [dict or None (default)] If provided, will set the plot-  
  ting style of the medians
- **meanprops** [dict or None (default)] If provided, will set the plot-  
  ting style of the means
- **meanline** [bool, default = False] If True (and **showmeans** is  
  True), will try to render the mean as a line spanning the  
  full width of the box according to **meanprops**. Not recom-  
  mended if **shownotches** is also True. Otherwise, means  
  will be shown as points.

**Returns result**: dict

A dictionary mapping each component of the boxplot to a list of the  
**matplotlib.lines.Line2D** instances created. That dictionary has the following keys (assuming vertical boxplots):

- **boxes**: the main body of the boxplot showing the quartiles  
  and the median’s confidence intervals if enabled.
- **medians**: horizontal lines at the median of each box.
- **whiskers**: the vertical lines extending to the most extreme,  
  n-outlier data points.
- **caps**: the horizontal lines at the ends of the whiskers.
- **fliers**: points representing data that extend beyond the  
  whiskers (outliers).
- **means**: points or lines representing the means.
Examples

- Default
- showmeans=True
- showmeans=True, meanline=True
- Tufte Style (showbox=False, showcaps=False)
- notch=True, bootstrap=10000
- showfliers=False
I never said they’d be pretty

Custom boxprops

Custom medianprops

and flierprops

whis="range"

Custom mean

as point

Custom mean

as line

whis=[15, 85]

#percentiles

broken_barh (xranges, yrange, **kwargs)

Plot horizontal bars.

Call signature:

broken_barh (self, xranges, yrange, **kwargs)

A collection of horizontal bars spanning yrange with a sequence of xranges.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xranges</td>
<td>sequence of (xmin, xwidth)</td>
</tr>
<tr>
<td>yrange</td>
<td>sequence of (ymin, ywidth)</td>
</tr>
</tbody>
</table>

kwargs are matplotlib.collections.BrokenBarHCollection properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.11 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

these can either be a single argument, i.e.:

\[
\text{facecolors} = \text{'black'}
\]

or a sequence of arguments for the various bars, i.e.:

\[
\text{facecolors} = \text{('black', 'red', 'green')}
\]
Example:

```python
bxp(bxpstats, positions=None, widths=None, vert=True, patch_artist=False, shownotches=False, showmeans=False, showcaps=True, showbox=True, showfliers=True, boxprops=None, whiskerprops=None, flierprops=None, medianprops=None, capprops=None, meanprops=None, meanline=False, manage_xticks=True)
```

Make a box and whisker plot for each column of \( x \) or each vector in sequence \( x \). The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

**Parameters bxpstats** : list of dicts
A list of dictionaries containing stats for each boxplot. Required keys are:

- **med**: The median (scalar float).
• q1: The first quartile (25th percentile) (scalar float).
• q3: The first quartile (50th percentile) (scalar float).
• whislo: Lower bound of the lower whisker (scalar float).
• whishi: Upper bound of the upper whisker (scalar float).

Optional keys are:
• mean: The mean (scalar float). Needed if showmeans=True.
• fliers: Data beyond the whiskers (sequence of floats). Needed if showfliers=True.
• cihi & cilow: Lower and upper confidence intervals about the median. Needed if shownotches=True.
• label: Name of the dataset (string). If available, this will be used as a tick label for the boxplot.

positions : array-like, default = [1, 2, ..., n]
Sets the positions of the boxes. The ticks and limits are automatically set to match the positions.

widths : array-like, default = 0.5
Either a scalar or a vector and sets the width of each box. The default is 0.5, or 0.15*(distance between extreme positions) if that is smaller.

vert : bool, default = False
If True (default), makes the boxes vertical. If False, makes horizontal boxes.

patch_artist : bool, default = False
If False produces boxes with the Line2D artist. If True produces boxes with the Patch artist.

shownotches : bool, default = False
If False (default), produces a rectangular box plot. If True, will produce a notched box plot.

showmeans : bool, default = False
If True, will toggle one the rendering of the means.

showcaps : bool, default = True
If True, will toggle one the rendering of the caps.

showbox : bool, default = True
If True, will toggle one the rendering of box.

showfliers : bool, default = True
If True, will toggle one the rendering of the fliers.

boxprops : dict or None (default)
If provided, will set the plotting style of the boxes.

whiskerprops : dict or None (default)
If provided, will set the plotting style of the whiskers.

capprops : dict or None (default)
If provided, will set the plotting style of the caps.

flierprops : dict or None (default)
If provided, will set the plotting style of the fliers.

medianprops : dict or None (default)
If provided, will set the plotting style of the medians.

meanprops : dict or None (default)
If provided, will set the plotting style of the means

**meanline**: bool, default = False

If True (and **showmeans** is True), will try to render the mean as a line spanning the full width of the box according to mean-props. Not recommended if **shownotches** is also True. Otherwise, means will be shown as points.

**manage_xticks**: bool, default = True

If the function should adjust the xlim and xtick locations.

**Returns result**: dict

A dictionary mapping each component of the boxplot to a list of the `matplotlib.lines.Line2D` instances created. That dictionary has the following keys (assuming vertical boxplots):

- **boxes**: the main body of the boxplot showing the quartiles and the median’s confidence intervals if enabled.
- **medians**: horizontal lines at the median of each box.
- **whiskers**: the vertical lines extending to the most extreme, n-outlier data points.
- **caps**: the horizontal lines at the ends of the whiskers.
- **fliers**: points representing data that extend beyond the whiskers (fliers).
- **means**: points or lines representing the means.
Examples

Default
showmeans=True
meanline=True

Tufte Style
(showbox=False,
showcaps=False)

notch=True
showfliers=False
I never said they'd be pretty

**Custom boxprops**

**Custom medianprops**

**Custom flierprops**

**Custom mean**

**as point**

**as line**

---

**can_pan()**

Return *True* if this axes supports any pan/zoom button functionality.

**can_zoom()**

Return *True* if this axes supports the zoom box button functionality.

**cla()**

Clear the current axes.

**clabel(CS, *args, **kwargs)**

Label a contour plot.

Call signature:

```python
clabel(cs, **kwargs)
```

Adds labels to line contours in `cs`, where `cs` is a `ContourSet` object returned by contour.
clabel(cs, v, **kwargs)

only labels contours listed in v.

Optional keyword arguments:

- **fontsize**: size in points or relative size e.g., ‘smaller’, ‘x-large’
- **colors**: if None, the color of each label matches the color of the corresponding contour
  - if one string color, e.g., colors = ‘r’ or colors = ‘red’, all labels will be plotted in this color
  - if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

- **inline**: controls whether the underlying contour is removed or not. Default is True.
- **inline_spacing**: space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.
- **fmt**: a format string for the label. Default is ‘%1.3f’ Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., fmt[level]=string), or it can be any callable, such as a Formatter instance, that returns a string when called with a numeric contour level.
- **manual**: if True, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location).

  manual can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.

- **rightside_up**: if True (default), label rotations will always be plus or minus 90 degrees from level.
- **use_clabeltext**: if True (default is False), ClabelText class (instead of matplotlib.Text) is used to create labels. ClabelText recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.
Simplest default with labels
**Single color - negative contours dashed**
clear()

clear the axes

cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x7f8bf872f320>,
window=<function window_hanning at 0x7f8bf872f050>, noverlap=0, pad_to=None,
sides='default', scale_by_freq=None, **kwargs)

Plot the coherence between x and y.

Call signature:

cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none,
window = mlab.window_hanning, noverlap=0, pad_to=None,
sides='default', scale_by_freq=None, **kwargs)

Plot the coherence between x and y. Coherence is the normalized cross spectral density:

\[ C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \]  

(39.1)

Keyword arguments:

Fs: scalar The sampling frequency (samples per time unit). It is used to calculate
the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

window: callable or ndarray A function or a vector of length NFFT. To create
window vectors see window_hanning(), window_none(),
numpy.blackman(), numpy.hamming(), numpy.bartlett().
scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides`: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

`pad_to`: integer The number of points to which the data segment is padded when performing the FFT. This can be different from NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to NFFT.

`NFFT`: integer The number of data points used in each block for the FFT. A power of 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

`detrend`: [‘default’ | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’] or callable
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

`scale_by_freq`: boolean
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

`noverlap`: integer The number of points of overlap between blocks. The default value is 0 (no overlap).

`Fc`: integer The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

The return value is a tuple (Cxy, f), where f are the frequencies of the coherence vector.

kwargs are applied to the lines.

References:

kwargs control the Line2D properties of the coherence plot:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
contains(mouseevent)

Test whether the mouse event occurred in the axes.

Returns True / False, {}.

contains_point(point)

Returns True if the point (tuple of x,y) is inside the axes (the area defined by the its patch). A pixel coordinate is required.

contour(*args, **kwargs)

Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

contourf() differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to contour().

Call signatures:

contour(Z)

make a contour plot of an array Z. The level values are chosen automatically.

contour(X,Y,Z)
$X, Y$ specify the $(x, y)$ coordinates of the surface

```python
contour(Z,N)
contour(X,Y,Z,N)
```

contour $N$ automatically-chosen levels.

```python
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence $V$

```python
contourf(..., V)
```

fill the $len(V)-1$ regions between the values in $V$

```python
contour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

$X$ and $Y$ must both be 2-D with the same shape as $Z$, or they must both be 1-D such that $len(X)$ is the number of columns in $Z$ and $len(Y)$ is the number of rows in $Z$.

```python
C = contour(...) returns a QuadContourSet object.
```

Optional keyword arguments:

- **colors**: `[None | string | (mpl_colors)]` If `None`, the colormap specified by `cmap` will be used.

  - If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

  - If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- **alpha**: `float` The alpha blending value

- **cmap**: `[None | Colormap]` A `cm Colormap` instance or `None`. If `cmap` is `None` and `colors` is `None`, a default Colormap is used.

- **norm**: `[None | Normalize]` A `matplotlib.colors.Normalize` instance for scaling data values to colors. If `norm` is `None` and `colors` is `None`, the default linear scaling is used.

- **vmin, vmax**: `[None | scalar]` If not `None`, either or both of these values will be supplied to the `matplotlib.colors.Normalize` instance, overriding the default color scaling based on `levels`.

- **levels**: `[level0, level1, ..., leveln]` A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass `levels=[0]`

- **origin**: `[None | ‘upper’ | ‘lower’ | ‘image’]` If `None`, the first value of $Z$ will correspond to the lower left corner, location $(0,0)$. If ‘image’, the rc value for `image.origin` will be used.

  - This keyword is not active if $X$ and $Y$ are specified in the call to `contour`.

- **extent**: `[None | (x0,x1,y0,y1)]`
If `origin` is not `None`, then `extent` is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of $Z[0,0]$ is the center of the pixel, not a corner. If `origin` is `None`, then $(x0, y0)$ is the position of $Z[0,0]$, and $(x1, y1)$ is the position of $Z[-1,-1]$.

This keyword is not active if $X$ and $Y$ are specified in the call to contour.

`locator`: [ `None` | `ticker.Locator subclass` ] If `locator` is `None`, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

`extend`: [ `‘neither’` | `‘both’` | `‘min’` | `‘max’` ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

`xunits`, `yunits`: [ `None` | `registered units` ] Override axis units by specifying an instance of `matplotlib.units.ConversionInterface`.

`antialiased`: [ `True` | `False` ] enable antialiasing, overriding the defaults. For filled contours, the default is `True`. For line contours, it is taken from `rcParams[‘lines.antialiased’]`.

contour-only keyword arguments:

`linestyles`: [ `None` | `number` | `tuple of numbers` ] If `linestyles` is `None`, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc contour.negative_linestyle` setting.

`linestyles` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

`nchunk`: [ `0` | `integer` ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly `nchunk` by `nchunk` points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless `antialiased` is `False`.

`hatches`: A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: contourf fills intervals that are closed at the top; that is, for boundaries $z1$ and $z2$, the filled region is:
$z_1 < z \leq z_2$

There is one exception: if the lowest boundary coincides with the minimum value of the $z$ array, then that minimum value will be included in the lowest interval.

Examples:
labels at selected locations

818 Chapter 39. axes
Crazy lines

39.1. matplotlib.axes
contourf(*args, **kwargs)
Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

contourf() differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to contour().

Call signatures:

contour(Z)

make a contour plot of an array Z. The level values are chosen automatically.

contour(X,Y,Z)

X, Y specify the (x, y) coordinates of the surface

contour(Z,N)
contour(X,Y,Z,N)

contour N automatically-chosen levels.
contour(Z,V)
contour(X,Y,Z,V)

draw contour lines at the values specified in sequence $V$

contourf(..., V)

fill the $\text{len}(V)-1$ regions between the values in $V$

contour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

$X$ and $Y$ must both be 2-D with the same shape as $Z$, or they must both be 1-D such that $\text{len}(X)$ is the number of columns in $Z$ and $\text{len}(Y)$ is the number of rows in $Z$.

$C = \text{contour}(...) \text{ returns a QuadContourSet object.}$

Optional keyword arguments:

colors: [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

alpha: float The alpha blending value

cmap: [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

norm: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

vmin, vmax: [None | scalar] If not None, either or both of these values will be supplied to the matplotlib.colors.Normalize instance, overriding the default color scaling based on levels.

levels: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass levels=[0]

origin: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of $Z$ will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if $X$ and $Y$ are specified in the call to contour.

extent: [None | (x0,x1,y0,y1)]

If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of $Z[0,0]$ is the center of the pixel, not a corner. If origin is None, then $(x0, y0)$ is the position of $Z[0,0]$, and $(x1, y1)$ is the position of $Z[-1,-1]$. 
This keyword is not active if X and Y are specified in the call to contour.

**locator:** [ *None* | ticker.Locator subclass ] If `locator` is *None*, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

**extend:** [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

**xunits, yunits:** [ *None* | registered units ] Override axis units by specifying an instance of `matplotlib.units.ConversionInterface`.

**antialiased:** [ *True* | *False* ] enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from rc-Params[‘lines.antialiased’].

Contour-only keyword arguments:

**linestyles:** [ *None* | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If `linestyles` is *None*, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc` `contour.negative_linestyle` setting.

`linestyles` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

Contourf-only keyword arguments:

**nchunk:** [ 0 | integer ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly `nchunk` by `nchunk` points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless `antialiased` is *False*.

**hatches:** A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: contourf fills intervals that are closed at the top; that is, for boundaries $z1$ and $z2$, the filled region is:

$$z1 < z \leq z2$$

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.
Examples:

Simplest default with labels
labels at selected locations
Single color - negative contours dashed
Nonsense (3 masked regions)

word length anomaly

sentence length anomaly

verbosity coefficient

Chapter 39. axes
Listed colors (3 masked regions)
**convert_xunits(x)**

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

**convert_yunits(y)**

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

**csd(x, y, NFFT=None, Fs=None, Fc=None, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=None, pad_to=None, sides='default', scale_by_freq=None, return_line=None, **kwargs)**

Plot the cross-spectral density.

Call signature:

\[
\text{csd}(x, y, NFFT=256, Fs=2, Fc=0, \text{detrend=mlab.detrend_none}, \\
\text{window=mlab.window_hanning}, \text{noverlap=0, pad_to=None}, \\
\text{sides='default', scale_by_freq=None, return_line=None, **kwargs})
\]

The cross spectral density \( P_{xy} \) by Welch’s average periodogram method. The vectors \( x \) and \( y \) are divided into \( NFFT \) length segments. Each segment is detrended by function \( \text{detrend} \) and windowed by function \( \text{window} \). \( \text{noverlap} \) gives the length of the overlap between segments. The product of the direct FFTs of \( x \) and \( y \) are averaged over each segment to compute \( P_{xy} \), with a scaling to correct for power loss due to windowing.

If \( \text{len}(x) < NFFT \) or \( \text{len}(y) < NFFT \), they will be zero padded to \( NFFT \).

\( x, y \): 1-D arrays or sequences Arrays or sequences containing the data

Keyword arguments:
**Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**window**: callable or ndarray A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**: ['default' | ‘onesided’ | ‘twosided’ ] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to**: integer The number of points to which the data segment is padded when performing the FFT. This can be different from `NFFT`, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `n` parameter in the call to `fft()`. The default is None, which sets `pad_to` equal to `NFFT`.

**NFFT**: integer The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use `pad_to` for this instead.

**detrend**: ['default' | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’] or callable The function applied to each segment before `fft-ing`, designed to remove the mean or linear trend. Unlike in MATLAB, where the `detrend` parameter is a vector, in matplotlib it is a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call `detrend_mean()`. ‘linear’ calls `detrend_linear()`. ‘none’ calls `detrend_none()`.

**scale_by_freq**: boolean Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**: integer The number of points of overlap between segments. The default value is 0 (no overlap).

**Fc**: integer The center frequency of `x` (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downs sampled to baseband.

**return_line**: bool Whether to include the line object plotted in the returned values. Default is False.

If `return_line` is False, returns the tuple `(Pxy, freqs)`. If `return_line` is True, returns the tuple `(Pxy, freqs, line)`.

**Pxy**: 1-D array The values for the cross spectrum \(P_{xy}\) before scaling (complex valued)
**freqs**: 1-D array The frequencies corresponding to the elements in $P_{xy}$

**line**: a **Line2D instance** The line created by this function. Only returned if `return_line` is True.

For plotting, the power is plotted as $10 \log_{10}(P_{xy})$ for decibels, though $P_{xy}$ itself is returned.


kwarg controls the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiasing or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.13 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

See also:

`psd()` psd() is the equivalent to setting y=x.

`drag_pan(button, key, x, y)`
Called when the mouse moves during a pan operation.

`button` is the mouse button number:
- 1: LEFT
- 2: MIDDLE
- 3: RIGHT

`key` is a “shift” key

`x, y` are the mouse coordinates in display coords.
Note: Intended to be overridden by new projection types.

```
draw (artist, renderer, *args, **kwargs)
Draw everything (plot lines, axes, labels)
draw_artist (a)
This method can only be used after an initial draw which caches the renderer. It is used to
efficiently update Axes data (axis ticks, labels, etc are not updated)
end_pan ()
Called when a pan operation completes (when the mouse button is up.)
```

Note: Intended to be overridden by new projection types.

```
errorbar (x, y, yerr=None, xerr=None, fmt='-', ecolor=None, elinewidth=None, capsize=3,
barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, **kwargs)
Plot an errorbar graph.
Call signature:

```
errorbar (x, y, yerr=None, xerr=None, fmt='-', ecolor=None, elinewidth=None, capsize=3,
barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, **kwargs)
```

Plot x versus y with error deltas in yerr and xerr. Vertical errorbars are plotted if yerr is not
None. Horizontal errorbars are plotted if xerr is not None.

x, y, xerr, and yerr can all be scalars, which plots a single error bar at x, y.

Optional keyword arguments:

- xerr/yerr: [ scalar | N, Nx1, or 2xN array-like ] If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn at +/-value relative to the data.

- fmt: [ ‘ ‘ | ‘none’ | plot format string ] The plot format symbol. If fmt is ‘none’ (case-insensitive), only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example. Default is ‘’, an empty plot format string; properties are then identical to the defaults for plot().

- ecolor: [ None | mpl color ] A matplotlib color arg which gives the color the errorbar lines; if None, use the color of the line connecting the markers.

- elinewidth: scalar The linewidth of the errorbar lines. If None, use the linewidth.

- capsize: scalar The length of the error bar caps in points

- capthick: scalar An alias kwarg to markeredgewidth (a.k.a. - mew). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if mew or markeredgewidth
are given, then they will over-ride `capthick`. This may change in future releases.

`barsabove`: [ `True` | `False` ] if `True`, will plot the errorbars above the plot symbols. Default is below.

`lolims / uplims / xlolims / xuplims`: [ `False` | `True` ] These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. lims-arguments may be of the same type as `xerr` and `yerr`. To use limits with inverted axes, `set_xlim()` or `set_ylim()` must be called before `errorbar()`.

`errorevery`: integer subsamples the errorbars. e.g., if everyerror=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

All other keyword arguments are passed on to the plot command for the markers. For example, this code makes big red squares with thick green edges:

```python
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s',
         mfc='red', mec='green', ms=20, mew=4)
```

where `mfc`, `mec`, `ms` and `mew` are aliases for the longer property names, `markerfacecolor`, `markeredgecolor`, `markersize` and `markeredgewidth`.

valid kwargs for the marker properties are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[ True</td>
</tr>
<tr>
<td><code>antialiased or aa</code></td>
<td>[ True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[ True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ <code>(Path, Transform)</code></td>
</tr>
<tr>
<td><code>color or c</code></td>
<td>any <code>matplotlib.color</code></td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>[ 'butt'</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>[ 'miter'</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>[ 'default'</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>[ 'full'</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle or ls</code></td>
<td>[ '-'</td>
</tr>
<tr>
<td><code>linewidth or lw</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[ True</td>
</tr>
<tr>
<td><code>marker</code></td>
<td><code>A valid marker style</code></td>
</tr>
<tr>
<td><code>markeredgecolor or mec</code></td>
<td>any <code>matplotlib.color</code></td>
</tr>
</tbody>
</table>
Table 39.14 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Returns (plotline, caplines, barlinecols):

<table>
<thead>
<tr>
<th>plotline: <strong>Line2D</strong> instance</th>
<th>x, y plot markers and/or line</th>
</tr>
</thead>
<tbody>
<tr>
<td>caplines: list of <strong>error bar cap</strong> Line2D instances</td>
<td></td>
</tr>
<tr>
<td>barlinecols: list of <strong>LineCollection</strong> instances for the horizontal and vertical error ranges.</td>
<td></td>
</tr>
</tbody>
</table>

Example:
eventplot(positions, orientation=u'horizontal', lineoffsets=1, linelengths=1,
linewidths=None, colors=None, linestyles=u'solid', **kwargs)
Plot identical parallel lines at specific positions.

Call signature:

```
eventplot(positions, orientation='horizontal', lineoffsets=0,
    linelengths=1, linewidths=None, color=None, linestyles='solid')
```

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

**orientation**

- [[ ‘horizontal’ | ‘vertical’ ]] ‘horizontal’ : the lines will be vertical and arranged in rows
- ‘vertical’ : lines will be horizontal and arranged in columns

**lineoffsets** : A float or array-like containing floats.
**linelengths** : A float or array-like containing floats.
**linewidths** : A float or array-like containing floats.
**colors** must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed) or a list of such sequences

**linestyles**: ['solid' | 'dashed' | 'dashdot' | 'dotted'] or an array of these values

For linelengths, linewidths, colors, and linestyles, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as positions, and each value will be applied to the corresponding row or column in positions.

Returns a list of `matplotlib.collections.EventCollection` objects that were added.

kwargs are `LineCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>axes</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clipbox</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>segments</td>
<td>unknown</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.15 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>verts</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

```python
fill(*args, **kwargs)
```

Plot filled polygons.

Call signature:

```python
fill(*args, **kwargs)
```

`args` is a variable length argument, allowing for multiple x, y pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at x, y in blue:

```python
ax.fill(x, y, 'b')
```

An arbitrary number of x, y, `color` groups can be specified:
ax.fill(x1, y1, 'g', x2, y2, 'r')

Return value is a list of **Patch** instances that were added.

The same color strings that **plot()** supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and y along x, use **fill_between()**

The *closed* kwarg will close the polygon when *True* (default).

**kwargs** control the **Polygon** properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <strong>Axes</strong> instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <strong>matplotlib.transforms.Bbox</strong> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <strong>matplotlib.figure.Figure</strong> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/ '</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><strong>Transform</strong> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Example:**
fill_between(x, y1, y2=0, where=None, interpolate=False, **kwargs)
Make filled polygons between two curves.

Call signature:
fill_between(x, y1, y2=0, where=None, **kwargs)

Create a PolyCollection filling the regions between y1 and y2 where where==True
x : An N-length array of the x data
y1 : An N-length array (or scalar) of the y data
y2 : An N-length array (or scalar) of the y data
where : If None, default to fill between everywhere. If not None, it is an N-length
        numpy boolean array and the fill will only happen over the regions where
        where==True.
interpolate : If True, interpolate between the two lines to find the precise point of
             intersection. Otherwise, the start and end points of the filled region will only
             occur on explicit values in the x array.
kwargs : Keyword args passed on to the PolyCollection.
        kwargs control the Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>[True</td>
</tr>
<tr>
<td>lod</td>
<td>unknown</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between y1 and 0

between y1 and 1

between y1 and y2
fill between where

Now regions with y2>1 are masked
See also:

```
fill_betweenx() for filling between two sets of x-values
```

```
fill_betweenx(y, x1, x2=0, where=None, **kwargs)
```

Make filled polygons between two horizontal curves.

Call signature:

```
fill_betweenx(y, x1, x2=0, where=None, **kwargs)
```

Create a `PolyCollection` filling the regions between `x1` and `x2` where `where==True`

- `y`: An N-length array of the y data
- `x1`: An N-length array (or scalar) of the x data
- `x2`: An N-length array (or scalar) of the x data
- `where`: If `None`, default to fill between everywhere. If not `None`, it is a N length numpy boolean array and the fill will only happen over the regions where `where==True`
- `kwargs`: keyword args passed on to the `PolyCollection`

`kwargs` control the `Polygon` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.18 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ <code>/</code></td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with <code>%s</code> conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between $y_1$ and 0

between $y_1$ and 1

between $y_1$ and $y_2$

x

39.1. matplotlib.axes
See also:

`fill_between()` for filling between two sets of y-values

`findobj(match=None, include_self=True)`
Find artist objects.

Recursively find all `Artist` instances contained in self.

`match` can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If `include_self` is True (default), include self in the list to be checked for a match.

`format_coord(x, y)`
Return a format string formatting the x, y coord

`format_xdata(x)`
Return x string formatted. This function will use the attribute `self.fmt_xdata` if it is callable, else will fall back on the xaxis major formatter

`format_ydata(y)`
Return y string formatted. This function will use the `fmt_ydata` attribute if it is callable, else will fall back on the yaxis major formatter
get_adjustable()

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_anchor()

get_animated()
    Return the artist’s animated state

get_aspect()

get_autoscale_on()
    Get whether autoscaling is applied for both axes on plot commands

get_autoscalex_on()
    Get whether autoscaling for the x-axis is applied on plot commands

get_autoscaley_on()
    Get whether autoscaling for the y-axis is applied on plot commands

get_axes()
    Return the Axes instance the artist resides in, or None

get_axes_locator()
    return axes_locator

get_axisbgcolor()
    Return the axis background color

get_axisbelow()
    Get whether axis below is true or not

get_children()
    return a list of child artists

get_clipbox()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_contains()
    Return the _contains test used by the artist, or None for default.

get_cursor_props()
    Return the cursor properties as a (linewidth, color) tuple, where linewidth is a float and color is an RGBA tuple
get_data_ratio()
   Returns the aspect ratio of the raw data.

   This method is intended to be overridden by new projection types.
get_data_ratio_log()
   Returns the aspect ratio of the raw data in log scale. Will be used when both axis scales are in log.
get_default_bbox_extra_artists()

get_figure()
   Return the Figure instance the artist belongs to.
get_frame_on()
   Get whether the axes rectangle patch is drawn
get_gid()
   Returns the group id
get_images()
   return a list of Axes images contained by the Axes
get_label()
   Get the label used for this artist in the legend.
get_legend()
   Return the legend.Legend instance, or None if no legend is defined
get_legend_handles_labels(legend_handler_map=None)
   Return handles and labels for legend

   ax.legend() is equivalent to

   h, l = ax.get_legend_handles_labels()
   ax.legend(h, l)
get_lines()
   Return a list of lines contained by the Axes
get_navigate()
   Get whether the axes responds to navigation commands
get_navigate_mode()
   Get the navigation toolbar button status: ‘PAN’, ‘ZOOM’, or None
get_path_effects()

get_picker()
   Return the picker object used by this artist
get_position(original=False)
   Return the a copy of the axes rectangle as a Bbox
get_rasterization_zorder()
    Get zorder value below which artists will be rasterized

get_rasterized()
    return True if the artist is to be rasterized

get_renderer_cache()

get_shared_x_axes()
    Return a copy of the shared axes Grouper object for x axes

get_shared_y_axes()
    Return a copy of the shared axes Grouper object for y axes

get_sketch_params()
    Returns the sketch parameters for the artist.
    Returns sketch_params : tuple or None
        A 3-tuple with the following elements:
        • scale: The amplitude of the wiggle perpendicular to the source line.
        • length: The length of the wiggle along the line.
        • randomness: The scale factor by which the length is shrunken or expanded.
        May return None if no sketch parameters were set.

get_snap()
    Returns the snap setting which may be:
    • True: snap vertices to the nearest pixel center
    • False: leave vertices as-is
    • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
    Only supported by the Agg and MacOSX backends.

get_tightbbox(renderer, call_axes_locator=True)
    Return the tight bounding box of the axes. The dimension of the Bbox in canvas coordinate.
    If call_axes_locator is False, it does not call the _axes_locator attribute, which is necessary to get the correct bounding box. call_axes_locator=False can be used if the caller is only interested in the relative size of the tightbbox compared to the axes bbox.

get_title(loc=u’center’)
    Get an axes title.
    Get one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.
    Parameters loc : {‘center’, ‘left’, ‘right’}, str, optional
        Which title to get, defaults to ‘center’
    Returns title: str
        The title text string.

get_transform()
    Return the Transform instance used by this artist.
get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining
    affine part of its transformation.

get_url()
    Returns the url

get_visible()
    Return the artist’s visibility

get_window_extent(*args, **kwargs)
    get the axes bounding box in display space; args and kwargs are empty

get_xaxis()
    Return the XAxis instance

get_xaxis_text1_transform(pad_points)
    Get the transformation used for drawing x-axis labels, which will add the given amount of
    padding (in points) between the axes and the label. The x-direction is in data coordinates and
    the y-direction is in axis coordinates. Returns a 3-tuple of the form:

    (transform, valign, halign)

    where valign and halign are requested alignments for the text.

    **Note:** This transformation is primarily used by the Axis class, and is meant to be overridden
    by new kinds of projections that may need to place axis elements in different locations.

get_xaxis_text2_transform(pad_points)
    Get the transformation used for drawing the secondary x-axis labels, which will add the given
    amount of padding (in points) between the axes and the label. The x-direction is in data
    coordinates and the y-direction is in axis coordinates. Returns a 3-tuple of the form:

    (transform, valign, halign)

    where valign and halign are requested alignments for the text.

    **Note:** This transformation is primarily used by the Axis class, and is meant to be overridden
    by new kinds of projections that may need to place axis elements in different locations.

get_xaxis_transform(\textit{which}=\textit{u’grid’})
    Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in
    data coordinates and the y-direction is in axis coordinates.

    **Note:** This transformation is primarily used by the Axis class, and is meant to be overridden
    by new kinds of projections that may need to place axis elements in different locations.

get_xbound()
    Returns the x-axis numerical bounds where:
lowerBound < upperBound

**get_xgridlines()**
Get the x grid lines as a list of Line2D instances

**get_xlabel()**
Get the xlabel text string.

**get_xlim()**
Get the x-axis range \([left, right]\)

**get_xmajorticklabels()**
Get the xtick labels as a list of Text instances.

**get_xminorticklabels()**
Get the x minor tick labels as a list of matplotlib.text.Text instances.

**get_xscale()**
Return the xaxis scale string: linear, log, symlog

**get_xticklabels**(*minor=False, which=None*)
Get the x tick labels as a list of Text instances.

**Parameters**

- minor : bool
  If True return the minor ticklabels, else return the major ticklabels

- which : None, \('\text{'minor'}, \text{'major'}, \text{'both'}\)
  Overrides minor.

**Returns**

- ret : list
  List of Text instances.

**get_xticklines()**
Get the xtick lines as a list of Line2D instances

**get_xticks**(*minor=False*)
Return the x ticks as a list of locations

**get_yaxis()**
Return the YAxis instance

**get_yaxis_text1_transform**(*pad_points*)
Get the transformation used for drawing y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

(transform, valign, halign)

where **valign** and **halign** are requested alignments for the text.

**Note:** This transformation is primarily used by the Axis class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.
get_yaxis_text2_transform (pad_points)
Get the transformation used for drawing the secondary y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

(transform, valign, halign)

where valign and halign are requested alignments for the text.

**Note:** This transformation is primarily used by the Axis class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_yaxis_transform (which='grid')
Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

**Note:** This transformation is primarily used by the Axis class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_ybound()
Return y-axis numerical bounds in the form of lowerBound < upperBound

get_ygridlines()
Get the y grid lines as a list of Line2D instances

get_ylabel()
Get the ylabel text string.

get_ylim()
Get the y-axis range [bottom, top]

get_ymajorticklabels()
Get the major y tick labels as a list of Text instances.

get_yminorticklabels()
Get the minor y tick labels as a list of Text instances.

get_yscale()
Return the yaxis scale string: linear, log, symlog

get_yticklabels (minor=False, which=None)
Get the x tick labels as a list of Text instances.

**Parameters**

- minor : bool
  If True return the minor ticklabels, else return the major ticklabels

- which : None, (‘minor’, ‘major’, ‘both’)
  Overrides minor.

**Returns**

- ret : list
  List of Text instances.
get_yticklines()
Get the ytick lines as a list of Line2D instances

get_yticks(minor=False)
Return the y ticks as a list of locations

get_zorder()
Return the Artist’s zorder.

grid(b=None, which='major', axis='both', **kwargs)
Turn the axes grids on or off.

Call signature:

grid(self, b=None, which='major', axis='both', **kwargs)

Set the axes grids on or off; b is a boolean. (For MATLAB compatibility, b may also be a string, ‘on’ or ‘off’.)

If b is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed that you want a grid and b is thus set to True.

which can be ‘major’ (default), ‘minor’, or ‘both’ to control whether major tick grids, minor tick grids, or both are affected.

axis can be ‘both’ (default), ‘x’, or ‘y’ to control which set of gridlines are drawn.

kwargs are used to set the grid line properties, e.g.:

ax.grid(color='r', linestyle='--', linewidth=2)

Valid Line2D kwars are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butts'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-', '--', '-.', ':', 'None', ' ' ]</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or me</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

### has_data()

Return `True` if any artists have been added to axes.

This should not be used to determine whether the `dataLim` need to be updated, and may not actually be useful for anything.

### have_units()

Return `True` if units are set on the x or y axes

### hexbin

```python
hexbin(x, y, C=None, gridsize=100, bins=None, xscale='linear', yscale='linear', extent=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors='none', reduce_C_function=<function mean at 0x7f8bfbc90230>, mincnt=None, marginals=False, **kwargs)
```

Make a hexagonal binning plot.

Call signature:

```python
hexbin(x, y, C=None, gridsize=100, bins=None, xscale='linear', yscale='linear',
       cmap=None, norm=None, vmin=None, vmax=None, alpha=None,
       linewidths=None, edgecolors='none', reduce_C_function=<function mean at 0x7f8bfbc90230>, mincnt=None, marginals=False, **kwargs)
```
alpha=None, linewidths=None, edgecolors='none'
reduce_C_function = np.mean, mincnt=None, marginals=True
**kwargs)

Make a hexagonal binning plot of x versus y, where x, y are 1-D sequences of the same length, N. If C is None (the default), this is a histogram of the number of occurences of the observations at (x[i],y[i]).

If C is specified, it specifies values at the coordinate (x[i],y[i]). These values are accumulated for each hexagonal bin and then reduced according to reduce_C_function, which defaults to numpy’s mean function (np.mean). (If C is specified, it must also be a 1-D sequence of the same length as x and y.)

x, y and/or C may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

- **gridsize**: [100 | integer] The number of hexagons in the x-direction, default is 100. The corresponding number of hexagons in the y-direction is chosen such that the hexagons are approximately regular. Alternatively, gridsize can be a tuple with two elements specifying the number of hexagons in the x-direction and the y-direction.

- **bins**: [None | 'log' | integer | sequence] If None, no binning is applied; the color of each hexagon directly corresponds to its count value.
  - If 'log', use a logarithmic scale for the color map. Internally, \(\log_{10}(i + 1)\) is used to determine the hexagon color.
  - If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.
  - If a sequence of values, the values of the lower bound of the bins to be used.

- **xscale**: ['linear' | 'log'] Use a linear or log10 scale on the horizontal axis.

- **yscale**: ['linear' | 'log'] Use a linear or log10 scale on the vertical axis.

- **mincnt**: [None | a positive integer] If not None, only display cells with more than mincnt number of points in the cell.

- **marginals**: [True | False] if marginals is True, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis.

- **extent**: [None | scalars (left, right, bottom, top)] The limits of the bins. The default assigns the limits based on gridsize, x, y, xscale and yscale.

Other keyword arguments controlling color mapping and normalization arguments:

- **cmap**: [None | Colormap] a matplotlib.colors.Colormap instance. If None, defaults to rc image.cmap.

- **norm**: [None | Normalize] matplotlib.colors.Normalize instance is used to scale luminance data to 0,1.

- **vmin / vmax**: scalar vmin and vmax are used in conjunction with norm to normalize luminance data. If either are None, the min and max of the color array C is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

- **alpha**: scalar between 0 and 1, or None the alpha value for the patches

- **linewidths**: [None | scalar] If None, defaults to rc lines.linewidth. Note that this is a tuple, and if you set thelinewidths argument you must set it as a sequence of floats, as required by RegularPolyCollection.

Other keyword arguments controlling the Collection properties:
edgecolors: [None | 'none' | mpl color | color sequence] If 'none', draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.

If None, draws the outlines in the default color.

If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the Collection kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[Path, Transform]</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’, ‘dashed’, ‘dashdot’, ‘dotted’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
The return value is a `PolyCollection` instance; use `get_array()` on this `PolyCollection` to get the counts in each hexagon. If `marginals` is True, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes `hbar` and `vbar`.

Example:

```python
hist(x, bins=10, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, **kwargs)
```

Plot a histogram.

Compute and draw the histogram of `x`. The return value is a tuple `(n, bins, patches)` or `([n0, n1, ...], bins, [patches0, patches1, ...])` if the input contains multiple data.

Multiple data can be provided via `x` as a list of datasets of potentially different length `([x0, x1, ...], bins, [patches0, patches1, ...])`, or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form.

Masked arrays are not supported at present.

**Parameters**
- `x` : (n,) array or sequence of (n,) arrays
  - Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length
- `bins` : integer or array_like, optional
  - If an integer is given, `bins + 1` bin edges are returned, consistently with `numpy.histogram()` for numpy version >= 1.3.
Unequally spaced bins are supported if `bins` is a sequence.

default is 10

**range** : tuple or None, optional
The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, `range` is `(x.min(), x.max())`. Range has no effect if `bins` is a sequence.

If `bins` is a sequence or `range` is specified, autoscaling is based on the specified bin range instead of the range of `x`.

Default is None

**normed** : boolean, optional
If True, the first element of the return tuple will be the counts normalized to form a probability density, i.e., `n/len(x)*dbin` i.e., the integral of the histogram will sum to 1. If stacked is also True, the sum of the histograms is normalized to 1.

Default is False

**weights** : (n,) array_like or None, optional
An array of weights, of the same shape as `x`. Each value in `x` only contributes its associated weight towards the bin count (instead of 1). If `normed` is True, the weights are normalized, so that the integral of the density over the range remains 1.

Default is None

**cumulative** : boolean, optional
If True, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If normed is also True then the histogram is normalized such that the last bin equals 1. If cumulative evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if `normed` is also True, then the histogram is normalized such that the first bin equals 1.

Default is False

**bottom** : array_like, scalar, or None
Location of the bottom baseline of each bin. If a scalar, the base line for each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

Default is None

**histtype** : {'bar', 'barstacked', 'step', 'stepfilled'}, optional
The type of histogram to draw.
- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are aranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
• 'stepfilled' generates a lineplot that is by default filled.

Default is 'bar'

**align** : {'left', 'mid', 'right'}, optional
Controls how the histogram is plotted.
- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

Default is 'mid'

**orientation** : {'horizontal', 'vertical'}, optional
If 'horizontal', `barh` will be used for bar-type histograms and the `bottom` kwarg will be the left edges.

**rwidth** : scalar or None, optional
The relative width of the bars as a fraction of the bin width. If None, automatically compute the width.

Ignored if `histtype` is 'step' or 'stepfilled'.

Default is None

**log** : boolean, optional
If True, the histogram axis will be set to a log scale. If log is True and x is a 1D array, empty bins will be filtered out and only the non-empty (n, bins, patches) will be returned.

Default is False

**color** : color or array_like of colors or None, optional
Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

Default is None

**label** : string or None, optional
String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

default is None

**stacked** : boolean, optional
If True, multiple data are stacked on top of each other If False multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

Default is False

**Returns**

- **n** : array or list of arrays
The values of the histogram bins. See `normed` and `weights` for a description of the possible semantics. If input x is an array, then this is an array of length `nbins`. If input is a sequence arrays [data1, data2, ...], then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

- **bins** : array
The edges of the bins. Length `nbins` + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple
data sets are passed in.

**patches**: list or list of lists

Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

**Other Parameters kwargs**: Patch properties

See also:

hist2d 2D histograms

**Notes**

Until numpy release 1.5, the underlying numpy histogram function was incorrect with `normed=True` if bin sizes were unequal. MPL inherited that error. It is now corrected within MPL when using earlier numpy versions.

**Examples**

```python
hist2d(x, y, bins=10, range=None, normed=False, weights=None, cmin=None, cmax=None, **kwargs)
```

Make a 2D histogram plot.

**Parameters x, y**: array_like, shape (n, )

Input values
bins: [None | int | [int, int] | array_like | [array, array]]
   The bin specification:
   • If int, the number of bins for the two dimensions
     (nx=ny=bins).
   • If [int, int], the number of bins in each dimension (nx, ny
     = bins).
   • If array_like, the bin edges for the two dimensions
     (x_edges=y_edges=bins).
   • If [array, array], the bin edges in each dimension
     (x_edges, y_edges = bins).
   The default value is 10.
range : array_like shape(2, 2), optional, default: None
   The leftmost and rightmost edges of the bins along each dimen-
   sion (if not specified explicitly in the bins parameters): [[xmin,
   xmax], [ymin, ymax]]. All values outside of this range will be
   considered outliers and not tallied in the histogram.
normed : boolean, optional, default: False
   Normalize histogram.
weights : array_like, shape (n,), optional, default: None
   An array of values w_i weighing each sample (x_i, y_i).
cmin : scalar, optional, default: None
   All bins that has count less than cmin will not be displayed and
   these count values in the return value count histogram will also
   be set to nan upon return
cmax : scalar, optional, default: None
   All bins that has count more than cmax will not be displayed
   (set to none before passing to imshow) and these count values
   in the return value count histogram will also be set to nan upon
   return

Returns
The return value is (counts, xedges, yedges, Image).

Other Parameters kwargs : pcolorfast() properties.

See also:
hist 1D histogram

Notes

Rendering the histogram with a logarithmic color scale is accomplished by passing a
colors.LogNorm instance to the norm keyword argument. Likewise, power-law normalization
(similar in effect to gamma correction) can be accomplished with colors.PowerNorm.
Examples

hitlist(event)
List the children of the artist which contain the mouse event event.

hlines(y, xmin, xmax, colors='k', linestyles='solid', label='', **kwargs)
Plot horizontal lines at each y from xmin to xmax.

Parameters
- y : scalar or sequence of scalar
  y-indexes where to plot the lines.
- xmin, xmax : scalar or 1D array_like
  Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- colors : array_like of colors, optional, default: ‘k’
- linestyles : [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’], optional
- label : string, optional, default: ‘’

Returns lines : LineCollection

See also:
- vlines vertical lines
Examples

Vertical lines demo

Horizontal lines demo

**hold**(

Call signature:

```python
hold(b=None)
```

Set the hold state. If *hold* is *None* (default), toggle the *hold* state. Else set the *hold* state to boolean value *b*.

Examples:

```python
# toggle hold
hold()

# turn hold on
hold(True)

# turn hold off
hold(False)
```

When *hold* is *True*, subsequent plot commands will be added to the current axes. When *hold* is *False*, the current axes and figure will be cleared on the next plot command.

**imshow**(*X*, *cmap=None*, *norm=None*, *aspect=None*, *interpolation=None*, *alpha=None*, *vmin=None*, *vmax=None*, *origin=None*, *extent=None*, *shape=None*, *filternorm=1*, *filt\-terrad=4.0*, *imlim=None*, *resample=None*, *url=None*, **kwargs)

Display an image on the axes.

**Parameters**

* \(X\) : array_like, shape (n, m) or (n, m, 3) or (n, m, 4)  
  Display the image in \(X\) to current axes. \(X\) may be a float array, a uint8 array or a PIL image. If \(X\) is an array, it can have the following shapes:
  - \(MxN\) – luminance (grayscale, float array only)
  - \(MxNx3\) – RGB (float or uint8 array)
•MxNx4 – RGBA (float or uint8 array)
  The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

**cmap** : Colormap, optional, default: None
  If None, default to rc image.cmap value. cmap is ignored when X has RGBA information

**aspect** : ['auto' | 'equal' | scalar], optional, default: None
  If ‘auto’, changes the image aspect ratio to match that of the axes.
  If ‘equal’, and extent is None, changes the axes aspect ratio to match that of the image. If extent is not None, the axes aspect ratio is changed to match that of the extent.
  If None, default to rc image.aspect value.

**interpolation** : string, optional, default: None
  If interpolation is None, default to rc image.interpolation. See also the filternorm and filterrad parameters. If interpolation is ‘none’, then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to ‘nearest’.

**norm** : Normalize, optional, default: None
  A Normalize instance is used to scale luminance data to 0, 1.
  If None, use the default func:normalize. norm is only used if X is an array of floats.

**vmin, vmax** : scalar, optional, default: None
  vmin and vmax are used in conjunction with norm to normalize luminance data. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

**alpha** : scalar, optional, default: None
  The alpha blending value, between 0 (transparent) and 1 (opaque)

**origin** : ['upper' | 'lower'], optional, default: None
  Place the [0,0] index of the array in the upper left or lower left corner of the axes. If None, default to rc image.origin.

**extent** : scalars (left, right, bottom, top), optional, default: None
  The location, in data-coordinates, of the lower-left and upper-right corners. If None, the image is positioned such that the pixel centers fall on zero-based (row, column) indices.

**shape** : scalars (columns, rows), optional, default: None
  For raw buffer images

**filternorm** : scalar, optional, default: 1
  A parameter for the antigrain image resize filter. From the anti-
grain documentation, if `filternorm = 1`, the filter normalizes integer values and corrects the rounding errors. It doesn’t do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

`filterrad` : scalar, optional, default: 4.0
  The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: ‘sinc’, ‘lanczos’ or ‘blackman’

Returns image : `AxesImage`

Other Parameters `kwargs` : `Artist` properties.

See also:

`matshow` Plot a matrix or an array as an image.

Examples

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.imshow(np.arange(16).reshape((4,4)), interpolation='nearest', origin='lower')
ax.set(xlim=(0, 4), ylim=(0, 4))
ax.invert_xaxis()
```

`in_axes(mouseevent)`
  Return `True` if the given `mouseevent` (in display coords) is in the Axes

`invert_xaxis()`
  Invert the x-axis.
invert_yaxis()
    Invert the y-axis.

is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

ishold()
    return the HOLD status of the axes

legend(*args, **kwargs)
    Places a legend on the axes.

    To make a legend for lines which already exist on the axes (via plot for instance), simply call
    this function with an iterable of strings, one for each legend item. For example:

    ```python
    ax.plot([1, 2, 3])
    ax.legend(["A simple line"])  
    ```

    However, in order to keep the “label” and the legend element instance together, it is preferable
    to specify the label either at artist creation, or by calling the `set_label()` method on the artist:

    ```python
    line, = ax.plot([1, 2, 3], label='Inline label')
    # Overwrite the label by calling the method.
    line.set_label("Label via method")
    ax.legend()
    ```

    Specific lines can be excluded from the automatic legend element selection by defining a label
    starting with an underscore. This is default for all artists, so calling `legend()` without any
    arguments and without setting the labels manually will result in no legend being drawn.

    For full control of which artists have a legend entry, it is possible to pass an iterable of legend
    artists followed by an iterable of legend labels respectively:

    ```python
    legend((line1, line2, line3), ("label1", "label2", "label3"))
    ```

    **Parameters**

    `loc` : int or string or pair of floats, default: 0
        The location of the legend. Possible codes are:

        | Location String | Location Code |
        |-----------------|---------------|
        | ‘best’          | 0             |
        | ‘upper right’   | 1             |
        | ‘upper left’    | 2             |
        | ‘lower left’    | 3             |
        | ‘lower right’   | 4             |
        | ‘right’         | 5             |
        | ‘center left’   | 6             |
        | ‘center right’  | 7             |
        | ‘lower center’  | 8             |
        | ‘upper center’  | 9             |
        | ‘center’        | 10            |
Alternatively can be a 2-tuple giving $x$, $y$ of the lower-left corner of the legend in axes coordinates (in which case `bbox_to_anchor` will be ignored).

**bbox_to_anchor**: `matplotlib.transforms.BboxBase` instance or tuple of floats

Specify any arbitrary location for the legend in `bbox_transform` coordinates (default Axes coordinates).

For example, to put the legend’s upper right hand corner in the center of the axes the following keywords can be used:

```python
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncol**: integer

The number of columns that the legend has. Default is 1.

**prop**: None or `matplotlib.font_manager.FontProperties` or dict

The font properties of the legend. If None (default), the current `matplotlib.rcParams` will be used.

**fontsize**: int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}

Controls the font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `prop` is not specified.

**numpoints**: None or int

The number of marker points in the legend when creating a legend entry for a line/`matplotlib.lines.Line2D`. Default is None which will take the value from the `legend.numpoints` rcParam.

**scatterpoints**: None or int

The number of marker points in the legend when creating a legend entry for a scatter plot/`matplotlib.collections.PathCollection`. Default is None which will take the value from the `legend.scatterpoints` rcParam.

**scatteryoff**: iterable of floats

The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5]. Default [0.375, 0.5, 0.3125].

**markerscale**: None or int or float

The relative size of legend markers compared with the originally drawn ones. Default is None which will take the value from the `legend.markerscale` rcParam.

**frameon**: None or bool

Control whether a frame should be drawn around the legend. Default is None which will take the value from the `legend.frameon` rcParam.

**fancybox**: None or bool
Control whether round edges should be enabled around the FancyBboxPatch which makes up the legend’s background. Default is None which will take the value from the legend.fancybox rcParam.

**shadow** : None or bool
Control whether to draw a shadow behind the legend. Default is None which will take the value from the legend.shadow rcParam.

**framealpha** : None or float
Control the alpha transparency of the legend’s frame. Default is None which will take the value from the legend.framealpha rcParam.

**mode** : {“expand”, None}
If mode is set to "expand" the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor if defines the legend’s size).

**bbox_transform** : None or matplotlib.transforms.Transform
The transform for the bounding box (bbox_to_anchor). For a value of None (default) the Axes’ transAxes transform will be used.

**title** : str or None
The legend’s title. Default is no title (None).

**borderpad** : float or None
The fractional whitespace inside the legend border. Measured in font-size units. Default is None which will take the value from the legend.borderpad rcParam.

**labelspacing** : float or None
The vertical space between the legend entries. Measured in font-size units. Default is None which will take the value from the legend.labelspacing rcParam.

**handlelength** : float or None
The length of the legend handles. Measured in font-size units. Default is None which will take the value from the legend.handlelength rcParam.

**handletextpad** : float or None
The pad between the legend handle and text. Measured in font-size units. Default is None which will take the value from the legend.handletextpad rcParam.

**borderaxespad** : float or None
The pad between the axes and legend border. Measured in font-size units. Default is None which will take the value from the legend.borderaxespad rcParam.

**columnspacing** : float or None
The spacing between columns. Measured in font-size units. Default is None which will take the value from the legend.columnspacing rcParam.

**handler_map** : dict or None
The custom dictionary mapping instances or
types to a legend handler. This `handler_map` updates the default handler map found at `matplotlib.legend.Legend.get_legend_handler_map()`.

**Notes**

Not all kinds of artist are supported by the legend command. See *Legend guide* for details.

**Examples**

```
locator_params((axis='both', tight=None, **kwargs))
```

Control behavior of tick locators.

Keyword arguments:
- `axis` [‘x’ | ‘y’ | ‘both’] Axis on which to operate; default is ‘both’.
- `tight` [True | False | None] Parameter passed to `autoscale_view()`. Default is None, for no change.

Remaining keyword arguments are passed to directly to the `set_params()` method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:
ax.locator_params(tight=True, nbins=4)

Because the locator is involved in autoscaling, autoscale_view() is called automatically after the parameters are changed.

This presently works only for the MaxNLocator used by default on linear axes, but it may be generalized.

**loglog(**args, **kwargs)**

Make a plot with log scaling on both the x and y axis.

Call signature:

`loglog(**kwargs)`


Notable keyword arguments:

- **base**/base: scalar > 1 Base of the x/y logarithm
- **sub**/sub: [None | sequence] The location of the minor x/y ticks; None defaults to autosubs, which depend on the number of decades in the plot; see `matplotlib.axes.Axes.set_xscale()` / `matplotlib.axes.Axes.set_yscale()` for details
- **nonposx**/nonposy: ['mask' | 'clip'] Non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
</tbody>
</table>
Table 39.21 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[’butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[’miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
**magnitude_spectrum**\( (x, Fs=\text{None}, Fc=\text{None}, \text{window}=\text{None}, \text{pad}_\text{to}=\text{None}, \text{sides}=\text{None}, \text{scale}=\text{None}, **\text{kwargs}) \)

Plot the magnitude spectrum.

Call signature:

```
magnitude_spectrum(x, Fs=2, Fc=0, \text{window}=\text{mlab.window_hanning}, 
\text{pad}_\text{to}=\text{None}, \text{sides}=\text{'default'}, **\text{kwargs})
```

Compute the magnitude spectrum of \(x\). Data is padded to a length of \(\text{pad}_\text{to}\) and the windowing function \(\text{window}\) is applied to the signal.

- **\text{x}: 1-D array or sequence** Array or sequence containing the data
- **Keyword arguments:**
  - **\text{Fs}: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
  - **\text{window}: callable or ndarry** A function or a vector of length \(\text{NFFT}\). To create window vectors see \text{window_hanning()}, \text{window_none()}, \text{numpy.blackman()}, \text{numpy.hamming()}, \text{numpy.bartlett()}, \text{scipy.signal()}, \text{scipy.signal.get_window()}, \text{etc}. The default is \text{window_hanning()}. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
  - **\text{sides}: [ ‘default’ | ‘onesided’ | ‘twosided’ ]** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real
data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to**: integer
The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

**scale**: ['default', 'linear', 'dB'] The scaling of the values in the spec. ‘linear’ is no scaling. ‘dB’ returns the values in dB scale. When mode is ‘density’, this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). ‘default’ is ‘linear’.

**Fc**: integer The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple (spectrum, freqs, line):

**spectrum**: 1-D array The values for the magnitude spectrum before scaling (real valued)

**freqs**: 1-D array The frequencies corresponding to the elements in spectrum

**line**: a Line2D instance The line created by this function

kwargs control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.22 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
See also:

\texttt{psd()} \texttt{psd()} plots the power spectral density.'.
\texttt{angle_spectrum()} \texttt{angle_spectrum()} plots the angles of the corresponding frequencies.
\texttt{phase_spectrum()} \texttt{phase_spectrum()} plots the phase (unwrapped angle) of the corresponding frequencies.
\texttt{specgram()} \texttt{specgram()} can plot the magnitude spectrum of segments within the signal in a colormap.

\texttt{margins(*args, **kw)}
Set or retrieve autoscaling margins.

signatures:

\texttt{margins()}

returns xmargin, ymargin

\texttt{margins(margin)}

\texttt{margins(xmargin, ymargin)}

\texttt{margins(x=xmargin, y=ymargin)}

\texttt{margins(..., tight=False)}
All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. The tight parameter is passed to autoscale_view(), which is executed after a margin is changed; the default here is True, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting tight to None will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if xmargin is not None, then xmargin times the X data interval will be added to each end of that interval before it is used in autoscaling.

\texttt{matshow(Z, **kwargs)}

Plot a matrix or array as an image.

The matrix will be shown the way it would be printed, with the first row at the top. Row and column numbering is zero-based.

\textbf{Parameters} \texttt{Z} : array_like shape (n, m)

The matrix to be displayed.

\textbf{Returns} \texttt{image} : AxesImage

\textbf{Other Parameters} \texttt{kwargs} : imshow arguments

Sets \texttt{origin} to ‘upper’, ‘interpolation’ to ‘nearest’ and ‘aspect’ to equal.

\textbf{See also:}

\texttt{imshow} plot an image
**minorticks_off()**
Remove minor ticks from the axes.

**minorticks_on()**
Add autoscaling minor ticks to the axes.

**name = u’rectilinear’**

**pchanged()**
Fire an event when property changed, calling all of the registered callbacks.

**pcolor(*args, **kwargs)**
Create a pseudocolor plot of a 2-D array.

**Note:** pcolor can be very slow for large arrays; consider using the similar but much faster **pcolormesh()** instead.

Call signatures:

- `pcolor(C, **kwargs)`
- `pcolor(X, Y, C, **kwargs)`

C is the array of color values.
X and Y, if given, specify the (x, y) coordinates of the colored quadrilaterals; the quadrilateral for $C[i,j]$ has corners at:

$$(X[i, j], Y[i, j]), (X[i, j+1], Y[i, j+1]), (X[i+1, j], Y[i+1, j]), (X[i+1, j+1], Y[i+1, j+1]).$$

Ideally the dimensions of X and Y should be one greater than those of C; if the dimensions are the same, then the last row and column of C will be ignored.

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y; for details, see the Grid Orientation section below.

If either or both of X and Y are 1-D arrays or column vectors, they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

X, Y and C may be masked arrays. If either C[i, j], or one of the vertices surrounding C[i,j] (X or Y at [i, j], [i+1, j], [i, j+1],[i+1, j+1]) is masked, nothing is plotted.

Keyword arguments:

- **cmap**: [None | Colormap] A matplotlib.colors.Colormap instance. If None, use rc settings.
- **norm**: [None | Normalize] An matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize().
- **vmin/vmax**: [None | scalar] vmin and vmax are used in conjunction with norm to normalize luminance data. If either is None, it is autoscaled to the respective min or max of the color array C. If not None, vmin or vmax passed in here override any pre-existing values supplied in the norm instance.
- **shading**: [‘flat’ | ‘faceted’] If ‘faceted’, a black grid is drawn around each rectangle; if ‘flat’, edges are not drawn. Default is ‘flat’, contrary to MATLAB. This kwarg is deprecated; please use ‘edgecolors’ instead:
  - shading="flat" – edgecolors="none"
  - shading="faceted" – edgecolors="k"
- **edgecolors**: [None | ‘none’ | color | color sequence] If None, the rc setting is used by default.
  - If ’none’, edges will not be visible.

An mpl color or sequence of colors will set the edge color

- **alpha**: $0 <= \text{scalar} <= 1$ or None the alpha blending value
- **snap**: bool Whether to snap the mesh to pixel boundaries.

Return value is a matplotlib.collections.Collection instance. The grid orientation follows the MATLAB convention: an array C with shape (nrows, ncolumns) is plotted with the column number as X and the row number as Y, increasing up; hence it is plotted the way the array would be printed, except that the Y axis is reversed. That is, C is taken as $C*(y,x)$.

Similarly for meshgrid():

```python
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```
is equivalent to:

\[
X = \text{array}([[0, 1, 2, 3, 4],
                 [0, 1, 2, 3, 4],
                 [0, 1, 2, 3, 4]])
\]

\[
Y = \text{array}([[0, 0, 0, 0, 0],
                 [1, 1, 1, 1, 1],
                 [2, 2, 2, 2, 2]])
\]

so if you have:

\[
C = \text{rand}(\text{len}(x), \text{len}(y))
\]

then you need to transpose \( C \):

\[
\text{pcolor}(X, Y, C.T)
\]

or:

\[
\text{pcolor}(C.T)
\]

MATLAB `pcolor()` always discards the last row and column of \( C \), but matplotlib displays the last row and column if \( X \) and \( Y \) are not specified, or if \( X \) and \( Y \) have one more row and column than \( C \).

kwargs can be used to control the `PolyCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[‘/’</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.23 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq)]</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Note:** The default `antialiased` is False if the default `edgecolors` = "none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If `edgecolors` is not "none", then the default `antialiased` is taken from `rcParams[‘patch.antialiased’]`, which defaults to True. Stroking the edges may be preferred if `alpha` is 1, but will cause artifacts otherwise.

See also:

`pcolormesh()` For an explanation of the differences between pcolor and pcolormesh.

`pcolorfast`(*args, **kwargs)

Pseudocolor plot of a 2-D array

Experimental: this is a pcolor-type method that provides the fastest possible rendering with the Agg backend, and that can handle any quadrilateral grid. It supports only flat shading (no outlines), it lacks support for log scaling of the axes, and it does not have a pyplot wrapper.

Call signatures:

ax.pcolorfast(C, **kwargs)
ax.pcolorfast(xr, yr, C, **kwargs)
ax.pcolorfast(x, y, C, **kwargs)
ax.pcolorfast(X, Y, C, **kwargs)

C is the 2D array of color values corresponding to quadrilateral cells. Let (nr, nc) be its shape. C may be a masked array.
ax.pcolorfast(C, **kwargs) is equivalent to ax.pcolorfast([0,nc], [0,nr], C, **kwargs)

xr, yr specify the ranges of x and y corresponding to the rectangular region bounding C. If:

\[ x_r = [x0, x1] \]

and:

\[ y_r = [y0, y1] \]

then \( x \) goes from \( x0 \) to \( x1 \) as the second index of \( C \) goes from 0 to \( nc \), etc. \((x0, y0)\) is the outermost corner of cell \((0,0)\), and \((x1, y1)\) is the outermost corner of cell \((nr-1, nc-1)\). All cells are rectangles of the same size. This is the fastest version.

\( x, y \) are 1D arrays of length \( nc + 1 \) and \( nr + 1 \), respectively, giving the x and y boundaries of the cells. Hence the cells are rectangular but the grid may be nonuniform. The speed is intermediate. (The grid is checked, and if found to be uniform the fast version is used.)

\( X \) and \( Y \) are 2D arrays with shape \((nr + 1, nc + 1)\) that specify the (x,y) coordinates of the corners of the colored quadrilaterals; the quadrilateral for \( C[i,j] \) has corners at \((X[i,j], Y[i,j]), (X[i,j+1], Y[i,j+1]), (X[i+1,j], Y[i+1,j]), (X[i+1,j+1], Y[i+1,j+1])\). The cells need not be rectangular. This is the most general, but the slowest to render. It may produce faster and more compact output using ps, pdf, and svg backends, however.

Note that the the column index corresponds to the x-coordinate, and the row index corresponds to y; for details, see the “Grid Orientation” section below.

Optional keyword arguments:

- **cmap**: [None | Colormap] A matplotlib.colors.Colormap instance from cm. If None, use rc settings.
- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize()
- **vmin/vmax**: [None | scalar] vmin and vmax are used in conjunction with norm to normalize luminance data. If either are None, the min and max of the color array \( C \) is used. If you pass a norm instance, vmin and vmax will be None.
- **alpha**: 0 <= scalar <= 1 or None the alpha blending value

Return value is an image if a regular or rectangular grid is specified, and a QuadMesh collection in the general quadrilateral case.

pcolormesh(*args, **kwargs)

Plot a quadrilateral mesh.

Call signatures:

- pcolormesh(C)
- pcolormesh(X, Y, C)
- pcolormesh(C, **kwargs)

Create a pseudocolor plot of a 2-D array.
pcolormesh is similar to pcolor(), but uses a different mechanism and returns a different object; pcolor returns a PolyCollection but pcolormesh returns a QuadMesh. It is much faster, so it is almost always preferred for large arrays.

C may be a masked array, but X and Y may not. Masked array support is implemented via cmap and norm; in contrast, pcolor() simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

- **cmap**: [None] | Colormap | A matplotlib.colors.Colormap instance. If None, use rc settings.
- **norm**: [None] | Normalize | A matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize().
- **vmin/vmax**: [None] | scalar | vmin and vmax are used in conjunction with norm to normalize luminance data. If either is None, it is autoscaled to the respective min or max of the color array C. If not None, vmin or vmax passed in here override any pre-existing values supplied in the norm instance.
- **shading**: ['flat' | 'gouraud'] | ‘flat’ indicates a solid color for each quad. When ‘gouraud’, each quad will be Gouraud shaded. When gouraud shading, edge-colors is ignored.
- **edgecolors**: [None] | 'None' | 'face' | color | color sequence | If None, the rc setting is used by default.
- **alpha**: 0 <= scalar <= 1 or None | the alpha blending value

Return value is a matplotlib.collections.QuadMesh object.

kwargs can be used to control the matplotlib.collections.QuadMesh properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.24 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>nothing or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

`pcolor()` For an explanation of the grid orientation and the expansion of 1-D X and/or Y to 2-D arrays.

`phase_spectrum(x, Fs=2, Fc=0, window=None, pad_to=None, sides='default', **kwargs)`

Plot the phase spectrum.

Call signature:

```python
phase_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning, pad_to=None, sides='default', **kwargs)
```

Compute the phase spectrum (unwrapped angle spectrum) of x. Data is padded to a length of `pad_to` and the windowing function `window` is applied to the signal.

**x**: 1-D array or sequence Array or sequence containing the data

Keyword arguments:

**Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**window**: callable or ndarray A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default
is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**: `[ ‘default’ | ‘onesided’ | ‘twosided’ ]` Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to**: integer
The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to `fft()`. The default is `None`, which sets `pad_to` equal to the length of the input signal (i.e. no padding).

**Fc**: integer The center frequency of $x$ (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple `(spectrum, freqs, line)`:

- **spectrum**: 1-D array The values for the phase spectrum in radians (real valued)
- **freqs**: 1-D array The frequencies corresponding to the elements in `spectrum`
- **line**: a Line2D instance The line created by this function

**kwargs** control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>['miter'</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>[#default’</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>[#full’</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>[‘-‘</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>marker</code></td>
<td>A valid marker style</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.25 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
See also:

- `magnitude_spectrum()` plots the magnitudes of the corresponding frequencies.
- `angle_spectrum()` plots the wrapped version of this function.
- `specgram()` can plot the phase spectrum of segments within the signal in a colormap.

**pick(** *args**)

Call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

**pickable**

Return `True` if `Artist` is pickable.

**pie**(*x*, `explode=None`, `labels=None`, `colors=None`, `autopct=None`, `pctdistance=0.6`, `shadow=False`, `labeldistance=1.1`, `startangle=None`, `radius=None`, `counterclock=True`, `wedgeprops=None`, `textprops=None`)

Plot a pie chart.

Call signature:
Make a pie chart of array \( x \). The fractional area of each wedge is given by \( x / \text{sum}(x) \). If \( \text{sum}(x) \leq 1 \), then the values of \( x \) give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the x-axis.

Keyword arguments:

- **explode**: [None | len(x) sequence] If not None, is a len(x) array which specifies the fraction of the radius with which to offset each wedge.
- **colors**: [None | color sequence] A sequence of matplotlib color args through which the pie chart will cycle.
- **labels**: [None | len(x) sequence of strings] A sequence of strings providing the labels for each wedge.
- **autopct**: [None | format string | format function] If not None, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be fmt% pct. If it is a function, it will be called.
- **pctdistance**: scalar The ratio between the center of each pie slice and the start of the text generated by autopct. Ignored if autopct is None; default is 0.6.
- **labeldistance**: scalar The radial distance at which the pie labels are drawn.
- **shadow**: [False | True] Draw a shadow beneath the pie.
- **startangle**: [None | Offset angle] If not None, rotates the start of the pie chart by angle degrees counterclockwise from the x-axis.
- **radius**: [None | scalar] The radius of the pie, if radius is None it will be set to 1.
- **counterclock**: [False | True] Specify fractions direction, clockwise or counterclockwise.
- **wedgeprops**: [None | dict of key value pairs] Dict of arguments passed to the wedge objects making the pie. For example, you can pass in wedgeprops = {'linewidth': 3} to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default clip_on=False.
- **textprops**: [None | dict of key value pairs] Dict of arguments to pass to the text objects.

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. E.g.:

```python
figure(figsize=(8,8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

or:

```python
axes(aspect=1)
```

**Return value:** If autopct is None, return the tuple (patches, texts):
• *patches* is a sequence of `matplotlib.patches.Wedge` instances
• *texts* is a list of the label `matplotlib.text.Text` instances.

If *autopct* is not `None`, return the tuple (*patches*, *texts*, *autotexts*), where *patches* and *texts* are as above, and *autotexts* is a list of `Text` instances for the numeric labels.

```python
plot(*args, **kwargs)
```

Plot lines and/or markers to the `Axes`. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal:

```python
plot(x, y)  # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)  # plot y using x as index array 0..N-1
plot(y, 'r+')  # ditto, but with red plusses
```

If *x* and/or *y* is 2-dimensional, then the corresponding columns will be plotted.

An arbitrary number of *x*, *y*, *fmt* groups can be specified, as in:

```python
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different color specified by a ‘color cycle’. To change this behavior, you can edit the `axes.color_cycle` rcParam.

The following format string characters are accepted to control the line style or marker:
### Line styles and markers

<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-</code></td>
<td>solid line style</td>
</tr>
<tr>
<td><code>--</code></td>
<td>dashed line style</td>
</tr>
<tr>
<td><code>-.</code></td>
<td>dash-dot line style</td>
</tr>
<tr>
<td><code>:</code></td>
<td>dotted line style</td>
</tr>
<tr>
<td>','</td>
<td>point marker</td>
</tr>
<tr>
<td><code>'</code></td>
<td>pixel marker</td>
</tr>
<tr>
<td><code>o</code></td>
<td>circle marker</td>
</tr>
<tr>
<td><code>'v</code></td>
<td>triangle_down marker</td>
</tr>
<tr>
<td><code>'^</code></td>
<td>triangle_up marker</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>triangle_left marker</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>triangle_right marker</td>
</tr>
<tr>
<td><code>'1</code></td>
<td>tri_down marker</td>
</tr>
<tr>
<td><code>'2</code></td>
<td>tri_up marker</td>
</tr>
<tr>
<td><code>'3</code></td>
<td>tri_left marker</td>
</tr>
<tr>
<td><code>'4</code></td>
<td>tri_right marker</td>
</tr>
<tr>
<td><code>'s</code></td>
<td>square marker</td>
</tr>
<tr>
<td><code>'p</code></td>
<td>pentagon marker</td>
</tr>
<tr>
<td><code>'s</code></td>
<td>star marker</td>
</tr>
<tr>
<td><code>'h</code></td>
<td>hexagon1 marker</td>
</tr>
<tr>
<td><code>'H</code></td>
<td>hexagon2 marker</td>
</tr>
<tr>
<td>`'+'</td>
<td>plus marker</td>
</tr>
<tr>
<td><code>'x</code></td>
<td>x marker</td>
</tr>
<tr>
<td><code>'D</code></td>
<td>diamond marker</td>
</tr>
<tr>
<td><code>'d</code></td>
<td>thin_diamond marker</td>
</tr>
<tr>
<td>`'</td>
<td>`</td>
</tr>
<tr>
<td>`'_'</td>
<td>hline marker</td>
</tr>
</tbody>
</table>

### Colors

The following color abbreviations are supported:

<table>
<thead>
<tr>
<th>character</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'b</code></td>
<td>blue</td>
</tr>
<tr>
<td><code>'g</code></td>
<td>green</td>
</tr>
<tr>
<td><code>'r</code></td>
<td>red</td>
</tr>
<tr>
<td><code>'c</code></td>
<td>cyan</td>
</tr>
<tr>
<td><code>'m</code></td>
<td>magenta</td>
</tr>
<tr>
<td><code>'y</code></td>
<td>yellow</td>
</tr>
<tr>
<td><code>'k</code></td>
<td>black</td>
</tr>
<tr>
<td><code>'w</code></td>
<td>white</td>
</tr>
</tbody>
</table>

In addition, you can specify colors in many weird and wonderful ways, including full names (`'green'`), hex strings (`'#008000'`), RGB or RGBA tuples ((0, 1, 0, 1)) or grayscale intensities as a string (`'0.8'`). Of these, the string specifications can be used in place of a `fmt` group, but the tuple forms can be used only as `kwargs`.

Line styles and colors are combined in a single format string, as in `'bo'` for blue circles.

The `kwargs` can be used to set line properties (any property that has a `set_*` method). You can use this to set a line label (for auto legends), linewidth, anitialising, marker face color, etc. Here is an example:
```
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs', label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and markerfacecolor with:

```
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See Line2D for details.

The kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

kwangs scalex and scaley, if defined, are passed on to autoscale_view() to determine whether the x and y axes are autoscaled; the default is True.

```python
plot_date(x, y, fmt='o', tz=None, xdate=True, ydate=False, **kwargs)
```

Plot with data with dates.

Call signature:
```
plot_date(x, y, fmt='bo', tz=None, xdate=True, ydate=False, **kwargs)
```

Similar to the plot() command, except the x or y (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or y can be a sequence of dates represented as float days since 0001-01-01 UTC.

Keyword arguments:
- `fmt`: string The plot format string.
- `tz`: [None | timezone string | tzinfo instance] The time zone to use in labeling dates. If None, defaults to rc value.
- `xdate`: [True | False] If True, the x-axis will be labeled with dates.
- `ydate`: [False | True] If True, the y-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to plot_date() since plot_date() will set the default tick locator to matplotlib.dates.AutoDateLocator (if the tick locator is not already set to a matplotlib.dates.DateLocator instance) and the default tick formatter to matplotlib.dates.AutoDateFormatter (if the tick formatter is not already set to a matplotlib.dates DateFormatter instance).
Valid kwargs are **Line2D** properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**See also:**

39.1. matplotlib.axes
**dates** for helper functions

date2num(), num2date() and drange() for help on creating the required floating point dates.

**properties()**
return a dictionary mapping property name -> value for all Artist props

**psd()**

psd(x, NFFT=None, Fs=2, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, return_line=None, **kwargs)

Plot the power spectral density.

Call signature:

psd(x, NFFT=256, Fs=2, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, return_line=None, **kwargs)

The power spectral density $P_{xx}$ by Welch’s average periodogram method. The vector $x$ is divided into $NFFT$ length segments. Each segment is detrended by function $detrend$ and windowed by function $window$. $noverlap$ gives the length of the overlap between segments. The $|\text{ftt}(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$, with a scaling to correct for power loss due to windowing.

If len($x$) < $NFFT$, it will be zero padded to $NFFT$.

$x$: 1-D array or sequence Array or sequence containing the data

Keyword arguments:

- **Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
- **window**: callable or ndarray A function or a vector of length $NFFT$. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hanning(), numpy.hamming(), numpy.bartlett(), scipy.signal(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
- **sides**: ['default' | 'onesided' | 'twosided'] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.
- **noverlap**: integer The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to fft(). The default is None, which sets **noverlap** equal to $NFFT$
- **NFFT**: integer The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use **pad_to** for this instead.
- **detrend**: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or callable
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

scale_by_freq: boolean
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap: integer
The number of points of overlap between segments. The default value is 0 (no overlap).

Fc: integer
The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return_line: bool
Whether to include the line object plotted in the returned values. Default is False.

If return_line is False, returns the tuple (Pxx, freqs). If return_line is True, returns the tuple (Pxx, freqs, line):

Pxx: 1-D array
The values for the power spectrum P_{xx} before scaling (real valued)

freqs: 1-D array
The frequencies corresponding to the elements in Pxx

line: a Line2D instance
The line created by this function. Only returned if return_line is True.

For plotting, the power is plotted as \(10 \log_{10}(P_{xx})\) for decibels, though \(P_{xx}\) itself is returned.


kwargs control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Example:**
See also:

specgram() specgram() differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.
magnitude_spectrum() magnitude_spectrum() plots the magnitude spectrum.
csd() csd() plots the spectral density between two signals.
quiver(*args, **kw)
Plot a 2-D field of arrows.
call signatures:

quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)

Arguments:

X, Y: The x and y coordinates of the arrow locations (default is tail of arrow; see pivot kwarg)
U, V: Give the x and y components of the arrow vectors
C: An optional array used to map colors to the arrows

All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if len(X)
and \( \text{len}(Y) \) match the column and row dimensions of \( U \), then \( X \) and \( Y \) will be expanded with \texttt{numpy.meshgrid()}. 

\( U, V, C \) may be masked arrays, but masked \( X, Y \) are not supported at present.

Keyword arguments:

**units:** ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy'] Arrow units; the arrow dimensions except for length are in multiples of this unit.

- 'width' or 'height': the width or height of the axes
- 'dots' or 'inches': pixels or inches, based on the figure dpi
- 'x', 'y', or 'xy': \( X, Y \), or \( \sqrt{X^2+Y^2} \) data units

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width' or 'height', the arrow size increases with the width and height of the axes, respectively, when the the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

**angles:** ['uv' | 'xy' | array] With the default 'uv', the arrow axis aspect ratio is 1, so that if \( U==*V \) the orientation of the arrow on the plot is 45 degrees CCW from the horizontal axis (positive to the right). With 'xy', the arrow points from \((x,y)\) to \((x+u, y+v)\). Use this for plotting a gradient field, for example. Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the horizontal axis. Note: inverting a data axis will correspondingly invert the arrows only with angles='xy'.

**scale:** [None | float] Data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. If None, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the scale_units parameter.

**scale_units:** None, or any of the units options. For example, if scale_units is 'inches', scale is 2.0, and \((u,v) = (1,0)\), then the vector will be 0.5 inches long. If scale_units is 'width', then the vector will be half the width of the axes.

If scale_units is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with \( u \) and \( v \) having the same units as \( x \) and \( y \), use angles='xy', scale_units='xy', scale=1". 

**width:** Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

**headwidth:** scalar Head width as multiple of shaft width, default is 3

**headlength:** scalar Head length as multiple of shaft width, default is 5

**headaxislength:** scalar Head length at shaft intersection, default is 4.5

**minshaft:** scalar Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

**minlength:** scalar Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

**pivot:** ['tail' | 'middle' | 'tip'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot.

**color:** [color | color sequence] This is a synonym for the PolyCollection facecolor kwarg. If \( C \) has been set, color has no effect.
The defaults give a slightly swept-back arrow; to make the head a triangle, make `headaxislength` the same as `headlength`. To make the arrow more pointed, reduce `headwidth` or increase `headlength` and `headaxislength`. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

Linewidths and edgecolors can be used to customize the arrow outlines. Additional `PolyCollection` keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[ ‘/’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>linestyles</code> or <code>dashes</code></td>
<td>‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code> or <code>linewidths</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

```
quiverkey(*args, **kw)
```
Add a key to a quiver plot.

Call signature:

```python
quiverkey(Q, X, Y, U, label, **kw)
```

Arguments:
- **Q**: The Quiver instance returned by a call to `quiver`
- **X, Y**: The location of the key; additional explanation follows.
- **U**: The length of the key
- **label**: A string with the length and units of the key

Keyword arguments:
- **coordinates** = [‘axes’ | ‘figure’ | ‘data’ | ‘inches’] Coordinate system and units for X, Y: ‘axes’ and ‘figure’ are normalized coordinate systems with 0,0 in the lower left and 1,1 in the upper right; ‘data’ are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); ‘inches’ is position in the figure in inches, with 0,0 at the lower left corner.
- **color**: overrides face and edge colors from Q.
- **labelpos** = [‘N’ | ‘S’ | ‘E’ | ‘W’] Position the label above, below, to the right, to the left of the arrow, respectively.
- **labelsep**: Distance in inches between the arrow and the label. Default is 0.1
- **labelcolor**: defaults to default `Text` color.
- **fontproperties**: A dictionary with keyword arguments accepted by the `FontProperties` initializer: `family`, `style`, `variant`, `size`, `weight`

Any additional keyword arguments are used to override vector properties taken from Q.

The positioning of the key depends on X, Y, coordinates, and labelpos. If labelpos is ‘N’ or ‘S’, X, Y give the position of the middle of the key arrow. If labelpos is ‘E’, X, Y positions the head, and if labelpos is ‘W’, X, Y positions the tail; in either of these two cases, X, Y is somewhere in the middle of the arrow+label key object.

**redraw_in_frame()**

This method can only be used after an initial draw which caches the renderer. It is used to efficiently update Axes data (axis ticks, labels, etc are not updated)

**relim(visible_only=False)**

Recompute the data limits based on current artists. If you want to exclude invisible artists from the calculation, set `visible_only=True`

At present, `Collection` instances are not supported.

**remove()**

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.

Note: there is no support for removing the artist’s legend entry.
remove_callback(oid)
Remove a callback based on its id.

See also:
add_callback() For adding callbacks
reset_position() Make the original position the active position

scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None, 
vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, **kwargs)
Make a scatter plot of x vs y, where x and y are sequence like objects of the same lengths.

Parameters x, y : array_like, shape (n, )
Input data
s : scalar or array_like, shape (n,), optional, default: 20
size in points^2.
c : color or sequence of color, optional, default
can be a single color format string, or a sequence of color
specifications of length N, or a sequence of N numbers to be
mapped to colors using the cmap and norm specified via kwargs
(see below). Note that c should not be a single numeric RGB or
RGBA sequence because that is indistinguishable from an array
of values to be colormapped. c can be a 2-D array in which the
rows are RGB or RGBA, however.
marker : MarkerStyle, optional, default: 'o'
See markers for more information on the different styles of
markers scatter supports.
cmap : Colormap, optional, default: None
A Colormap instance or registered name. cmap is only used if
c is an array of floats. If None, defaults to rc image.cmap.
norm : Normalize, optional, default: None
A Normalize instance is used to scale luminance data to 0, 1.
norm is only used if c is an array of floats. If None, use the
default normalize().
vmin, vmax : scalar, optional, default: None
vmin and vmax are used in conjunction with norm to normalize
luminance data. If either are None, the min and max of the color
array is used. Note if you pass a norm instance, your settings
for vmin and vmax will be ignored.
alpha : scalar, optional, default: None
The alpha blending value, between 0 (transparent) and 1
(opaque)
linewidths : scalar or array_like, optional, default: None
If None, defaults to (lines.linewidth,). Note that this is a tuple,
and if you set the linewidths argument you must set it as a se-
quence of floats, as required by RegularPolyCollection.

Returns paths : PathCollection

Other Parameters kwargs : Collection properties
Notes

Any or all of $x$, $y$, $s$, and $c$ may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.

Examples

```
semilogx(*args, **kwargs)

Make a plot with log scaling on the $x$ axis.

Call signature:

    semilogx(*args, **kwargs)

    semilogx() supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_yscale()`.

Notable keyword arguments:

    `basex`: `scalar > 1` Base of the $x$ logarithm
    `subsx`: `[None | sequence]` The location of the minor xticks; `None` defaults to autosubs, which depend on the number of decades in the plot; see `set_yscale()` for details.
```
**nonposx:** [‘mask’ | ‘clip’] Non-positive values in x can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)]</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_jjoinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'.</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_jjoinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

39.1. matplotlib.axes
See also:

loglog() For example code and figure

semilogy(*args, **kwargs)
Make a plot with log scaling on the y axis.

call signature:

semilogy(*args, **kwargs)

semilogy() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

- **basey**: scalar > 1 Base of the y logarithm
- **subsy**: [None | sequence] The location of the minor yticks; None defaults to autosubs, which depend on the number of decades in the plot; see set_yscale() for details.
- **nonposy**: ['mask' | 'clip'] Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.pyplot.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgcolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39.31 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinalstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

loglog() For example code and figure

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_adjustable(adjustable)
ACCEPTS: [ ‘box’ | ‘datalim’ | ‘box-forced’]

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha value used for blending - not supported on all backends.

ACCEPTS: float (0.0 transparent through 1.0 opaque)

set_anchor(anchorn)
anchor
set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_aspect(aspect, adjustable=None, anchor=None)

<table>
<thead>
<tr>
<th>aspect</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘auto’</td>
<td>automatic: fill position rectangle with data</td>
</tr>
<tr>
<td>‘normal’</td>
<td>same as ‘auto’, deprecated</td>
</tr>
<tr>
<td>‘equal’</td>
<td>same scaling from data to plot units for x and y</td>
</tr>
<tr>
<td>num</td>
<td>a circle will be stretched such that the height is num times the width.</td>
</tr>
<tr>
<td></td>
<td>aspect=1 is the same as aspect=’equal’.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>adjustable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘box’</td>
<td>change physical size of axes</td>
</tr>
<tr>
<td>‘datalim’</td>
<td>change xlim or ylim</td>
</tr>
<tr>
<td>‘box-forced’</td>
<td>same as ‘box’, but axes can be shared</td>
</tr>
</tbody>
</table>

‘box’ does not allow axes sharing, as this can cause unintended side effect. For cases when sharing axes is fine, use ‘box-forced’.

anchor

<table>
<thead>
<tr>
<th>anchor</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘C’</td>
<td>centered</td>
</tr>
<tr>
<td>‘SW’</td>
<td>lower left corner</td>
</tr>
<tr>
<td>‘S’</td>
<td>middle of bottom edge</td>
</tr>
<tr>
<td>‘SE’</td>
<td>lower right corner</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

Deprecated since version 1.2: the option ‘normal’ for aspect is deprecated. Use ‘auto’ instead.

set_autoscale_on(b)
Set whether autoscaling is applied on plot commands

accepts: [ True | False ]

set_autoscalex_on(b)
Set whether autoscaling for the x-axis is applied on plot commands

accepts: [ True | False ]
**set_autoscaley_on**\( (b) \)
Set whether autoscaling for the y-axis is applied on plot commands

accepts: [ True | False ]

**set_axes**\( (axes) \)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

**set_axes_locator**\( (locator) \)
set axes_locator

ACCEPT: a callable object which takes an axes instance and renderer and returns a bbox.

**set_axis_bgcolor**\( (color) \)
set the axes background color

ACCEPTS: any matplotlib color - see colors()

**set_axis_off()**
turn off the axis

**set_axis_on()**
turn on the axis

**set_axisbelow\( (b) \)**
Set whether the axis ticks and gridlines are above or below most artists

ACCEPTS: [ True | False ]

**set_clip_box**\( (clipbox) \)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

**set_clip_on**\( (b) \)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**set_clip_path**\( (path, transform=None) \)
Set the artist’s clip path, which may be:

- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [ (Path, Transform) | Patch | None ]

**set_color_cycle**\( (clist) \)
Set the color cycle for any future plot commands on this Axes.

clist is a list of mpl color specifiers.
set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_cursor_props(*args)
Set the cursor property as:

ax.set_cursor_props(linewidth, color)

or:

ax.set_cursor_props((linewidth, color))

ACCEPTS: a (float, color) tuple

set_figure(fig)
Set the class: Axes figure
accepts a class: Figure instance

set_frame_on(b)
Set whether the axes rectangle patch is drawn

ACCEPTS: [True | False]

set_gid(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

set_lod(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

set_navigate(b)
Set whether the axes responds to navigation toolbar commands

ACCEPTS: [True | False]
**set_navigate_mode**(\(b\))
Set the navigation toolbar button status;

**Warning:** this is not a user-API function.

**set_path_effects**(\(path\_effects\))
set path_effects, which should be a list of instances of matplotlib.path._Base class or its derivatives.

**set_picker**(\(picker\))
Set the epsilon for picking used by this artist

picker can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_position**(\(pos, which='both'\))
Set the axes position with:

\[
\text{pos} = [\text{left, bottom, width, height}]
\]

in relative 0,1 coords, or \(pos\) can be a Bbox

There are two position variables: one which is ultimately used, but which may be modified by apply_aspect(), and a second which is the starting point for apply_aspect().

**Optional keyword arguments:** *which*

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'active'</td>
<td>to change the first</td>
</tr>
<tr>
<td>'original'</td>
<td>to change the second</td>
</tr>
<tr>
<td>'both'</td>
<td>to change both</td>
</tr>
</tbody>
</table>

**set_rasterization_zorder**(\(z\))
Set zorder value below which artists will be rasterized. Set to None to disable rasterizing of artists below a particular zorder.

**set_rasterized**(\(rasterized\))
Force rasterized (bitmap) drawing in vector backend output.
Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sketch_params** *(scale=None, length=None, randomness=None)*

Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  The scale factor by which the length is shrunk or expanded (default 16.0)

**set_snap**(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_title**(label, fontdict=None, loc=u’center’, **kwargs)

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

- **label**: str
  Text to use for the title

- **fontdict**: dict
  A dictionary controlling the appearance of the title text, the default fontdict is:

  ```
  {'fontsize': rcParams['axes.titlesize'],
   'fontweight': rcParams['axes.titleweight'],
   'verticalalignment': 'baseline',
   'horizontalalignment': 'center'}
  ```

- **loc**: {'center', 'left', 'right'}, str, optional
  Which title to set, defaults to ‘center’

**Returns**

- **text**: Text
  The matplotlib text instance representing the title

**Other Parameters**

- **kwargs**: text properties
  Other keyword arguments are text properties, see Text for a list of valid text properties.

**set_transform**(t)

Set the Transform instance used by this artist.

ACCEPTS: Transform instance
**set_url** *(url)*
Sets the url for the artist

**set_visible** *(b)*
Set the artist’s visibility.

**set_xbound** *(lower=None, upper=None)*
Set the lower and upper numerical bounds of the x-axis. This method will honor axes inversion regardless of parameter order. It will not change the _autoscaleXon attribute.

**set_xlabel** *(xlabel, fontdict=None, labelpad=None, **kwargs)*
Set the label for the x-axis.

**Parameters**

- **xlabel** : string
  x label
- **labelpad** : scalar, optional, default: None
  spacing in points between the label and the x-axis

**Other Parameters**

- **kwargs** : Text properties

**See also:**
- **text** for information on how override and the optional args work

**set_xlim** *(left=None, right=None, emit=True, auto=False, **kw)*
Call signature:

```
set_xlim(self, *args, **kwargs):
```

Set the data limits for the xaxis

**Examples:**

```
set_xlim((left, right))
set_xlim(left, right)
set_xlim(left=1)  # right unchanged
set_xlim(right=1) # left unchanged
```

**Keyword arguments:**

- **left** : scalar
  The left xlim; xmin, the previous name, may still be used
- **right** : scalar
  The right xlim; xmax, the previous name, may still be used
- **emit** : [ True | False ]
  Notify observers of limit change
- **auto** : [ True | False | None ]
  Turn x autoscaling on (True), off (False; default), or leave unchanged (None)

Note, the **left** (formerly **xmin**) value may be greater than the **right** (formerly **xmax**). For example, suppose x is years before present. Then one might use:

```
set_xlim(5000, 0)
```

so 5000 years ago is on the left of the plot and the present is on the right.

Returns the current xlimits as a length 2 tuple
ACCEPTEs: length 2 sequence of floats

**set_xmargin**(m)
Set padding of X data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscal-
ing.
accepts: float in range 0 to 1

**set_xscale**(value, **kwargs)
Call signature:

set_xscale(value)

Set the scaling of the x-axis: u’linear’ | u’log’ | u’symlog’

ACCEPTS: [u’linear’ | u’log’ | u’symlog’]

**Different kwargs are accepted, depending on the scale:** ‘linear’

‘log’

*baseX/baseY*: The base of the logarithm

*nonposX/nonposY*: [’mask’ | ‘clip’] non-positive values in x or y can be
masked as invalid, or clipped to a very small positive number

*subsx/subsy*: Where to place the subticks between each major tick. Should
be a sequence of integers. For example, in a log10 scale: [2, 3, 4,
5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major
tick.

‘symlog’

*baseX/baseY*: The base of the logarithm

*linthreshX/linthreshY*: The range (-x, x) within which the plot is linear (to
avoid having the plot go to infinity around zero).

*subsx/subsy*: Where to place the subticks between each major tick. Should
be a sequence of integers. For example, in a log10 scale: [2, 3, 4,
5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major
tick.

{linscaleX/linscaleY}: This allows the linear range (-linthresh to linthresh) to
be stretched relative to the logarithmic range. Its value is the num-
ber of decades to use for each half of the linear range. For example,
when linscale == 1.0 (the default), the space used for the positive and
negative halves of the linear range will be equal to one decade in the
logarithmic range.

**set_xticklabels**(labels, fontdict=None, minor=False, **kwargs)
Call signature:

set_xticklabels(labels, fontdict=None, minor=False, **kwargs)
Set the xtick labels with list of strings `labels`. Return a list of axis text instances.

`kwargs` set the Text properties. Valid properties are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>or fontname or name</td>
<td></td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment</td>
<td>[‘center’</td>
</tr>
<tr>
<td>or ha</td>
<td></td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>verticalalignment or ma</td>
<td>[‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
\texttt{set_xticks}(ticks, \textit{minor}=\textit{False})

Set the x ticks with list of \textit{ticks}

\textbf{ACCEPTS: sequence of floats}

\texttt{set_ybound}(\textit{lower}=None, \textit{upper}=None)

Set the lower and upper numerical bounds of the y-axis. This method will honor axes inversion regardless of parameter order. It will not change the \_autoscaleYon attribute.

\texttt{set_ylabel}(\textit{ylabel}, \textit{fontdict}=None, \textit{labelpad}=None, **\textit{kwargs})

Set the label for the yaxis

\begin{itemize}
  \item \textbf{Parameters} \textit{ylabel} : string
    \begin{itemize}
      \item \textit{y label}
    \end{itemize}
  \item \textit{labelpad} : scalar, optional, default: None
    \begin{itemize}
      \item spacing in points between the label and the x-axis
    \end{itemize}
  \end{itemize}

\textbf{Other Parameters} \textit{kwargs} : \texttt{Text} properties

\textbf{See also:}

\texttt{text} for information on how override and the optional args work

\texttt{set_ylim}(\textit{bottom}=None, \textit{top}=None, \textit{emit}=True, \textit{auto}=False, **\textit{kw})

Call signature:

\begin{verbatim}
set_ylim(self, *args, **kwargs):
\end{verbatim}

Set the data limits for the yaxis

Examples:

set_ylim((bottom, top))
set_ylim(bottom, top)
set_ylim(bottom=1) # top unchanged
set_ylim(top=1) # bottom unchanged

\begin{itemize}
  \item \textbf{Keyword arguments:}
    \begin{itemize}
      \item \textit{bottom} : scalar The bottom ylim; the previous name, \textit{ymin}, may still be used
      \item \textit{top} : scalar The top ylim; the previous name, \textit{ymax}, may still be used
      \item \textit{emit} : [ \textit{True} | \textit{False} ] Notify observers of limit change
      \item \textit{auto} : [ \textit{True} | \textit{False} | \textit{None} ] Turn y autoscaling on (\textit{True}), off (\textit{False}; default), or leave unchanged (\textit{None})
    \end{itemize}
\end{itemize}

Note, the \textit{bottom} (formerly \textit{ymin}) value may be greater than the \textit{top} (formerly \textit{ymax}). For example, suppose \textit{y} is depth in the ocean. Then one might use:

\begin{verbatim}
set_ylim(5000, 0)
\end{verbatim}

so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

Returns the current ylimits as a length 2 tuple

\textbf{ACCEPTS: length 2 sequence of floats}
**set_ymargin**($m$)

Set padding of Y data limits prior to autoscaling.

$m$ times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

**set_yscale**(value, **kwargs)

Call signature:

```
set_yscale(value)
```

Set the scaling of the y-axis: u’linear’ | u’log’ | u’symlog’

ACCEPTS: [u’linear’ | u’log’ | u’symlog’]

**Different kwargs are accepted, depending on the scale:**

- **linear**
  - `base2`/`base10`: The base of the logarithm
  - `nonposx`/`nonposy`: [‘mask’ | ‘clip’] non-positive values in $x$ or $y$ can be masked as invalid, or clipped to a very small positive number
  - `subsx`/`subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

- **log**
  - `base2`/`base10`: The base of the logarithm
  - `linthreshx`/`linthreshy`: The range (-$x$, $x$) within which the plot is linear (to avoid having the plot go to infinity around zero).
  - `subsx`/`subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.
  - `linscalex`/`linscaley`: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when linscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

**set_yticklabels**(labels, fontdict=None, minor=False, **kwargs)

Call signature:

```
set_yticklabels(labels, fontdict=None, minor=False, **kwargs)
```

Set the y tick labels with list of strings $labels$. Return a list of Text instances.
**kwargs can set Text properties for the labels. Valid properties are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>backgroundcolor</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>bbox</code></td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>family</code> or <code>fontfamily</code> or <code>fontname</code> or <code>name</code></td>
<td>[FONTNAME</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fontproperties</code> or <code>font_properties</code></td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>horizontalalignment</code> or <code>ha</code></td>
<td>[ 'center'</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linespacing</code></td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>multialignment</code></td>
<td>['left'</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>position</code></td>
<td>(x,y)</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>rotation</code></td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td><code>rotation_mode</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>size</code> or <code>fontsize</code></td>
<td>[size in points</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>stretch</code> or <code>fontstretch</code></td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td><code>style</code> or <code>fontstyle</code></td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><code>text</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>variant</code> or <code>fontvariant</code></td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><code>verticalalignment</code> or <code>va</code> or <code>ma</code></td>
<td>[ ‘center’</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>weight</code> or <code>fontweight</code></td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td><code>x</code></td>
<td>float</td>
</tr>
<tr>
<td><code>y</code></td>
<td>float</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPTS: sequence of strings
**set_yticks**(*ticks*, **minor**=`False`)  
Set the y ticks with list of *ticks*

ACCEPTS: sequence of floats

Keyword arguments:

- **minor**: `[False | True]` Sets the minor ticks if `True`

**set_zorder**(*level*)  
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**specgram**(*x*, **NFFT**=`None`, **Fs**=`None`, **Fc**=`None`, **detrend**=`None`, **window**=`None`, **noverlap**=`None`, **cmap**=`None`, **xextent**=`None`, **pad_to**=`None`, **sides**=`None`, **scale_by_freq**=`None`, **mode**=`None`, **scale**=`None`, **vmin**=`None`, **vmax**=`None`, **kwargs**)  
Plot a spectrogram.

Call signature:

```python
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=128, cmap=None, xextent=None, pad_to=None, sides='default', scale_by_freq=None, mode='default', scale='default', **kwargs)
```

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

**x**: 1-D array or sequence  
Array or sequence containing the data

Keyword arguments:

- **Fs**: scalar  
The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray  
A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: `[‘default’ | ‘onesided’ | ‘twosided’]`  
Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer  
The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail.
This corresponds to the \( n \) parameter in the call to \( \text{fft()}. \) The default is None, which sets \texttt{pad to} equal to \texttt{NFFT}.

**NFFT**: integer The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should \textit{NOT} be used to get zero padding, or the scaling of the result will be incorrect. Use \texttt{pad to} for this instead.

**detrend**: [ ‘default’ | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’] or callable

The function applied to each segment before \( \text{fft-} \)ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the \texttt{detrend} parameter is a vector, in matplotlib it is a function. The \texttt{pylab} module defines \texttt{detrend_none()}, \texttt{detrend_mean()}, and \texttt{detrend_linear()}, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call \texttt{detrend_mean()}. ‘linear’ calls \texttt{detrend_linear()}. ‘none’ calls \texttt{detrend_none()}.  

**scale_by_freq**: boolean

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz\(^{-1}\). This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**mode**: [ ‘default’ | ‘psd’ | ‘magnitude’ | ‘angle’ | ‘phase’ ] What sort of spectrum to use. Default is ‘psd’. which takes the power spectral density. ‘complex’ returns the complex-valued frequency spectrum. ‘magnitude’ returns the magnitude spectrum. ‘angle’ returns the phase spectrum without unwrapping. ‘phase’ returns the phase spectrum with unwrapping.

**noverlap**: integer The number of points of overlap between blocks. The default value is 128.

**scale**: [ ‘default’ | ‘linear’ | ‘dB’ ] The scaling of the values in the \texttt{spec}. ‘linear’ is no scaling. ‘dB’ returns the values in dB scale. When \texttt{mode} is ‘psd’, this is dB power \((10 \times \log_{10})\). Otherwise this is dB amplitude \((20 \times \log_{10})\). ‘default’ is ‘dB’ if \texttt{mode} is ‘psd’ or ‘magnitude’ and ‘linear’ otherwise. This must be ‘linear’ if \texttt{mode} is ‘angle’ or ‘phase’.

**Fc**: integer The center frequency of \( x \) (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**cmap**: A matplotlib.colors.Colormap instance; if \textit{None}, use default determined by \texttt{rc}

**xextent**: The image extent along the x-axis. \texttt{xextent = (xmin,xmax)} The default is \((0,max(bins))\), where \( bins \) is the return value from \texttt{specgram()}

**kwargs**: Additional kwargs are passed on to \texttt{imshow} which makes the \texttt{specgram} image

**Note**: \texttt{detrend} and \texttt{scale_by_freq} only apply when \texttt{mode} is set to ‘psd’

Returns the tuple \((\texttt{spectrum}, \texttt{freqs}, \texttt{t}, \texttt{im})\):

**spectrum**: 2-D array columns are the periodograms of successive segments

**freqs**: 1-D array The frequencies corresponding to the rows in \texttt{spectrum}

**t**: 1-D array The times corresponding to midpoints of segments (i.e the columns in \texttt{spectrum})

**im**: instance of class \texttt{AxesImage} The image created by \texttt{imshow} containing the
spectrogram

Example:

![Example Image]

See also:

- `psd()` differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.
- `magnitude_spectrum()` A single spectrum, similar to having a single segment when `mode` is 'magnitude'. Plots a line instead of a colormap.
- `angle_spectrum()` A single spectrum, similar to having a single segment when `mode` is 'angle'. Plots a line instead of a colormap.
- `phase_spectrum()` A single spectrum, similar to having a single segment when `mode` is 'phase'. Plots a line instead of a colormap.

```python
spy(Z, precision=0, marker=None, markersize=None, aspect='equal', origin='upper', **kwargs)
```

Plot the sparsity pattern on a 2-D array.

`spy(Z)` plots the sparsity pattern of the 2-D array `Z`.

**Parameters**

- `Z` : sparse array (n, m)
  - The array to be plotted.
- `precision` : float, optional, default: 0
  - If `precision` is 0, any non-zero value will be plotted; else, values of `|Z| > precision` will be plotted.

For scipy.sparse.spmatrix instances, there is a special
case: if $precision$ is ‘present’, any value present in the array
will be plotted, even if it is identically zero.

$origin$: ["upper", "lower"], optional, default: “upper”
Place the [0,0] index of the array in the upper left or lower left
corner of the axes.

$aspect$: ["auto" | "equal" | scalar], optional, default: “equal”
If ‘equal’, and $extent$ is None, changes the axes aspect ratio to
match that of the image. If $extent$ is not None, the axes aspect
ratio is changed to match that of the extent.

If ‘auto’, changes the image aspect ratio to match that of the
axes.

If None, default to rc image.aspect value.

Two plotting styles are available: image or marker. Both
are available for full arrays, but only the marker style
works for :class:`scipy.sparse.spmatrix` instances.

If *marker* and *markersize* are *None*, an image will be
returned and any remaining kwargs are passed to
:func:`matplotlib.pyplot.imshow`; else, a
:class:`matplotlib.lines.Line2D` object will be returned with
the value of marker determining the marker type, and any
remaining kwargs passed to the

If *marker* and *markersize* are *None*, useful kwargs include:

- **cmap**
- **alpha**

See also:

- `imshow` for image options.
- `plot` for plotting options

```
stackplot(x, *args, **kwargs)
```

Draws a stacked area plot.

$x$ : 1d array of dimension $N$

$y$ [2d array of dimension $M \times N$, OR any number 1d arrays each of dimension] $1 \times N$. The data is
assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)  # where $y$ is $M \times N$
stackplot(x, y1, y2, y3, y4)  # where $y1$, $y2$, $y3$, $y4$, are all $1 \times Nm$
```

Keyword arguments:
**baseline** [‘zero’, ‘sym’, ‘wiggle’, ‘weighted_wiggle’] Method used to calculate the baseline. ‘zero’ is just a simple stacked plot. ‘sym’ is symmetric around zero and is sometimes called ThemeRiver. ‘wiggle’ minimizes the sum of the squared slopes. ‘weighted_wiggle’ does the same but weights to account for size of each layer. It is also called Streamgraph-layout. More details can be found at http://www.leebyron.com/else/streamgraph/.

**colors** [A list or tuple of colors. These will be cycled through and] used to colour the stacked areas. All other keyword arguments are passed to `fill_between()`

Returns r: A list of `PolyCollection`, one for each element in the stacked area plot.

Note that `Legend` does not support `PolyCollection` objects. To create a legend on a stackplot, use a proxy artist: http://matplotlib.org/users/legend_guide.html#using-proxy-artist

### start_pan(x, y, button)

Called when a pan operation has started.

x, y are the mouse coordinates in display coords. button is the mouse button number:
- • 1: LEFT
- • 2: MIDDLE
- • 3: RIGHT

**Note:** Intended to be overridden by new projection types.

### stem(*args, **kwargs)

Create a stem plot.

Call signatures:

stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')
stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')

A stem plot plots vertical lines (using `linefmt`) at each x location from the baseline to y, and places a marker there using `markerfmt`. A horizontal line at 0 is is plotted using `basefmt`.

If no x values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (markerline, stemlines, baseline).

**See also:**

This document for details.

**Example:**
**step** *(x, y, *args, **kwargs)*

Make a step plot.

Call signature:

```python
step(x, y, *args, **kwargs)
```

Additional keyword args to **step()** are the same as those for **plot()**.

*x* and *y* must be 1-D sequences, and it is assumed, but not checked, that *x* is uniformly increasing.

Keyword arguments:

- **where**: [ ‘pre’ | ‘post’ | ‘mid’ ] If ‘pre’, the interval from *x*[i] to *x*[i+1] has level *y*[i+1]
  - If ‘post’, that interval has level *y*[i]
  - If ‘mid’, the jumps in *y* occur half-way between the *x*-values.

**streamplot** *(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None, arrowsize=1, arrowstyle='->', minlength=0.1, transform=None, zorder=1)*

Draws streamlines of a vector flow.

*x, y [1d arrays] an evenly spaced grid.*

*u, v [2d arrays] x and y-velocities. Number of rows should match length of y, and the number of columns should match x.*
**density** [float or 2-tuple] Controls the closeness of streamlines. When density = 1, the domain is divided into a 30x30 grid—density linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use [density_x, density_y].

**linewidth** [numeric or 2d array] Vary linewidth when given a 2d array with the same shape as velocities.

**color** [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, color values are converted to colors using cmap.

**cmap** [Colormap] Colormap used to plot streamlines and arrows. Only necessary when using an array input for color.

**norm** [Normalize] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when color is an array.

**arrowsize** [float] Factor scale arrow size.


**minlength** [float] Minimum length of streamline in axes coordinates.

**zorder** [int] Any number

Returns:

**stream_container** [StreamplotSet] Container object with attributes
- lines: matplotlib.collections.LineCollection of streamlines
- arrows: collection of matplotlib.patches.FancyArrowPatch objects representing arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

**table(** **kwargs** **)**

Add a table to the current axes.

Call signature:

```python
table(cellText=None, cellColours=None, cellLoc='right', colWidths=None, rowLabels=None, rowColours=None, rowLoc='left', colLabels=None, colColours=None, colLoc='center', loc='bottom', bbox=None):
```

Returns a matplotlib.table.Table instance. For finer grained control over tables, use the Table class and add it to the axes with add_table().

Thanks to John Gill for providing the class and table.

**kwargs** control the Table properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontsize</td>
<td>a float in points</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**text**(x, y, s, fontdict=None, withdash=False, **kwargs)

Add text to the axes.

Add text in string s to axis at location x, y, data coordinates.

**Parameters x, y** : scalars
data coordinates

s : string
text

fontdict : dictionary, optional, default: None
A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

withdash : boolean, optional, default: False
Creates a TextWithDash instance instead of a Text instance.

**Other Parameters** kwargs : Text properties.
Other miscellaneous text parameters.

**Examples**

Individual keyword arguments can be used to override any given parameter:

```python
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0.0 is lower-left and 1.1 is upper-right). The example below places text in the
center of the axes:

```python
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...       verticalalignment='center',
...       transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword bbox. bbox is a dictionary of Rectangle properties. For example:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

**tick_params** *(axis='both', **kwargs)*
Change the appearance of ticks and tick labels.

Keyword arguments:
- **axes** [ ['x' | 'y' | 'both']] Axis on which to operate; default is 'both'.
- **reset** [ [True | False]] If True, set all parameters to defaults before processing other keyword arguments. Default is False.
- **which** [ ['major' | 'minor' | 'both']] Default is 'major'; apply arguments to which ticks.
- **direction** [ ['in' | 'out' | 'inout']] Puts ticks inside the axes, outside the axes, or both.
- **length** Tick length in points.
- **width** Tick width in points.
- **color** Tick color; accepts any mpl color spec.
- **pad** Distance in points between tick and label.
- **labelsize** Tick label font size in points or as a string (e.g., 'large').
- **labelcolor** Tick label color; mpl color spec.
- **colors** Changes the tick color and the label color to the same value: mpl color spec.
- **zorder** Tick and label zorder.
- **bottom, top, left, right** [ [bool | 'on' | 'off']]] controls whether to draw the respective ticks.
- **labelbottom, labelsop, labelleft, labelright** Boolean or ['on' | 'off'], controls whether to draw the respective tick labels.

Example:

```python
ax.tick_params(direction='out', length=6, width=2, colors='r')
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red.

**ticklabel_format** (**kwargs**)
Change the ScalarFormatter used by default for linear axes.

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>style</code></td>
<td>['sci' (or 'scientific')</td>
</tr>
<tr>
<td><code>scilimits</code></td>
<td>(m, n), pair of integers; if <code>style</code> is 'sci', scientific notation will be used for numbers outside the range $10^m$ to $10^n$. Use (0,0) to include all numbers.</td>
</tr>
<tr>
<td><code>useOffset</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axis</code></td>
<td>['x'</td>
</tr>
</tbody>
</table>

Only the major ticks are affected. If the method is called when the `ScalarFormatter` is not the `Formatter` being used, an `AttributeError` will be raised.

**tricontour(*args, **kwargs)**

Draw contours on an unstructured triangular grid. `tricontour()` and `tricontourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```
tricontour(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)  
tricontour(x, y, triangles=triangles, ...)  
tricontour(x, y, mask=mask, ...)  
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining arguments may be:

```
tricontour(..., Z)
```

where Z is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, N)
```

contour N automatically-chosen levels.

```
tricontour(..., Z, V)
```
draw contour lines at the values specified in sequence $V$

$$\text{tricontourf}(..., Z, V)$$

fill the $(\text{len}(V)-1)$ regions between the values in $V$

$$\text{tricontour}(Z, \text{**kwargs})$$

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

$C = \text{tricontour}(\ldots)$ returns a TriContourSet object.

Optional keyword arguments:

- **colors** : [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.
  
  If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
  
  If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- **alpha** : float The alpha blending value

- **cmap** : [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

- **norm** : [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

- **levels** : [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass $\text{levels}=[0]$.

- **origin** : [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.
  
  This keyword is not active if $X$ and $Y$ are specified in the call to contour.

- **extent** : [None | (x0,x1,y0,y1)]

  If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(); it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If origin is None, then $(x0, y0)$ is the position of Z[0,0], and $(x1, y1)$ is the position of Z[-1,-1].
  
  This keyword is not active if $X$ and $Y$ are specified in the call to contour.

- **locator** : [None | ticker.Locator subclass] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the $V$ argument.

- **extend** : [‘neither’ | ‘both’ | ‘min’ | ‘max’] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range,
but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

`xunits, yunits`: [None | registered units] Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

tricontour-only keyword arguments:

`linewidths`: [None | number | tuple of numbers] If `linewidths` is None, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

`linestyles`: [None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’] If `linestyles` is None, the ‘solid’ is used.

`linestyles` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in `contour.negative_linestyle` in `matplotlibrc` will be used.

tricontourf-only keyword arguments:

`antialiased`: [True | False] enable antialiasing

`nchunk`: [0 | integer] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly `nchunk` by `nchunk` points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless `antialiased` is False.

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries $z_1$ and $z_2$, the filled region is:

$$z_1 < z \leq z_2$$

There is one exception: if the lowest boundary coincides with the minimum value of the $z$ array, then that minimum value will be included in the lowest interval.

`Examples:`
Contour plot of Delaunay triangulation
**tricontour**(*args, **kwargs)

Draw contours on an unstructured triangular grid. tricontour() and tricontourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

tricontour(triangulation, ...)

where triangulation is a matplotlib.tri.Triangulation object, or

tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)

in which case a Triangulation object will be created. See **Triangulation** for a explanation of these possibilities.

The remaining arguments may be:

tricontour(..., Z)
where Z is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, N)
```

contour N automatically-chosen levels.

```
tricontour(..., Z, V)
```

draw contour lines at the values specified in sequence V

```
tricontourf(..., Z, V)
```

fill the (len(V)-1) regions between the values in V

```
tricontour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

```
C = tricontour(...)
```

returns a TriContourSet object.

Optional keyword arguments:

- **colors**: [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.
  - If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
  - If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- **alpha**: float The alpha blending value

- **cmap**: [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

- **levels**: [level0, level1, ..., levelN] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass levels=[0]

- **origin**: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

  This keyword is not active if X and Y are specified in the call to contour.

- **extent**: [None | (x0,x1,y0,y1)]

  If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If origin is None, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1].

  This keyword is not active if X and Y are specified in the call to contour.
**locator**: [ *None* | *ticker.Locator subclass* ] If *locator* is *None*, the default *MaxNLocator* is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

**extend**: [ ‘*neither*’ | ‘*both*’ | ‘min’ | ‘max’ ] Unless this is ‘*neither*’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via *matplotlib.colors.Colormap.set_under()* and *matplotlib.colors.Colormap.set_over()* methods.

**xunits, yunits**: [ *None* | *registered units* ] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

tricontour-only keyword arguments:

**linewidths**: [ *None* | *number* | tuple of numbers ] If *linewidths* is *None*, the default width in *lines.linewidth* in *matplotlibrc* is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles**: [ *None* | ‘*solid*’ | ‘*dashed*’ | ‘*dashdot*’ | ‘*dotted*’ ] If *linestyles* is *None*, the ‘*solid*’ is used.

*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in *contour.negative_linestyle* in *matplotlibrc* will be used.

tricontourf-only keyword arguments:

**antialiased**: [ *True* | *False* ] enable antialiasing

**nchunk**: [ 0 | integer ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

\[ z1 < z \leq z2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

**Examples**:
Contour plot of Delaunay triangulation
**tripcolor(***args, **kwargs*)**

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```
tripcolor(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tripcolor(x, y, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The next argument must be `C`, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the kwarg `facecolors*=C instead of just *C`. 
shading may be ‘flat’ (the default) or ‘gouraud’. If shading is ‘flat’ and C values are defined at points, the color values used for each triangle are from the mean C of the triangle’s three points. If shading is ‘gouraud’ then color values must be defined at points. shading of ‘faceted’ is deprecated; please use edgecolors instead.

The remaining kwargs are the same as for pcolor().

Example:

```python
import matplotlib.pyplot as plt

X, Y = np.meshgrid(np.arange(-3, 3, 0.1), np.arange(-3, 3, 0.1))
Z = np.sqrt(X**2 + Y**2)

fig, ax = plt.subplots()
tripcolor = ax.tripcolor(X, Y, Z, shading='flat')
fig.colorbar(tripcolor)

plt.show()
```
**tripcolor of Delaunay triangulation, gouraud shading**

**tripcolor of user-specified triangulation**

```python
tripcolor(*args, **kwargs)
```

Draw a unstructured triangular grid as lines and/or markers.
The triangulation to plot can be specified in one of two ways; either:

\[ \text{triplot(triangulation, ...)} \]

where triangulation is a \texttt{matplotlib.tri.Triangulation} object, or

\[ \text{triplot(x, y, ...)} \]
\[ \text{triplot(x, y, triangles, ...)} \]
\[ \text{triplot(x, y, triangles=triangles, ...)} \]
\[ \text{triplot(x, y, mask=mask, ...)} \]
\[ \text{triplot(x, y, triangles, mask=mask, ...)} \]

in which case a Triangulation object will be created. See \texttt{Triangulation} for a explanation of these possibilities.

The remaining args and kwargs are the same as for \texttt{plot()}. Return a list of 2 \texttt{Line2D} containing respectively:
- the lines plotted for triangles edges
- the markers plotted for triangles nodes

\textbf{Example:}

```
triplot of Delaunay triangulation
```

![Triplot of Delaunay triangulation](image)
**twinx()**

Call signature:

```python
ax = twinx()
```

create a twin of Axes for generating a plot with a sharex x-axis but independent y axis. The y-axis of self will have ticks on left and the returned axes will have ticks on the right.

**Note:** For those who are ‘picking’ artists while using twinx, pick events are only called for the artists in the top-most axes.

**twiny()**

Call signature:

```python
ax = twiny()
```

create a twin of Axes for generating a plot with a shared y-axis but independent x axis. The x-axis of self will have ticks on bottom and the returned axes will have ticks on the top.

**Note:** For those who are ‘picking’ artists while using twiny, pick events are only called for the artists in the top-most axes.

**update**(props)

Update the properties of this **Artist** from the dictionary **prop**.
update_datalim(xys, updatex=True, updatey=True)
Update the data lim bbox with seq of xy tups or equiv. 2-D array

update_datalim_bounds(bounds)
Update the datalim to include the given Bbox bounds

update_datalim_numerix(x, y)
Update the data lim bbox with seq of xy tups

update_from(other)
Copy properties from other to self.

violin(vpstats, positions=None, vert=True, widths=0.5, showmeans=False, showextrema=True, showmedians=False)
Drawing function for violin plots.

Call signature:

violin(vpstats, positions=None, vert=True, widths=0.5,
       showmeans=False, showextrema=True, showmedians=False):

Draw a violin plot for each column of vpstats. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.

**Parameters vpstats**: list of dicts
A list of dictionaries containing stats for each violin plot. Required keys are:

- **coords**: A list of scalars containing the coordinates that the violin’s kernel density estimate were evaluated at.
- **vals**: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in coords.
- **mean**: The mean value for this violin’s dataset.
- **median**: The median value for this violin’s dataset.
- **min**: The minimum value for this violin’s dataset.
- **max**: The maximum value for this violin’s dataset.

**positions**: array-like, default = [1, 2, ..., n]
Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

**vert**: bool, default = True.
If true, plots the violins vertically. Otherwise, plots the violins horizontally.

**widths**: array-like, default = 0.5
Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

**showmeans**: bool, default = False
If true, will toggle rendering of the means.

**showextrema**: bool, default = True
If true, will toggle rendering of the extrema.

**showmedians**: bool, default = False
If true, will toggle rendering of the medians.
Returns result: dict
A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- **bodies**: A list of the `matplotlib.collections.PolyCollection` instances containing the filled area of each violin.
- **means**: A `matplotlib.collections.LineCollection` instance created to identify the mean values of each of the violin’s distribution.
- **mins**: A `matplotlib.collections.LineCollection` instance created to identify the bottom of each violin’s distribution.
- **maxes**: A `matplotlib.collections.LineCollection` instance created to identify the top of each violin’s distribution.
- **bars**: A `matplotlib.collections.LineCollection` instance created to identify the centers of each violin’s distribution.
- **medians**: A `matplotlib.collections.LineCollection` instance created to identify the median values of each of the violin’s distribution.

**violinplot***(dataset, positions=None, vert=True, widths=0.5, showmeans=False, showextrema=True, showmedians=False, points=100, bw_method=None)*

Make a violin plot.

Call signature:

```python
violinplot(dataset, positions=None, vert=True, widths=0.5, showmeans=False, showextrema=True, showmedians=False, points=100, bw_method=None):
```

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.

**Parameters**

- **dataset**: Array or a sequence of vectors.
  The input data.

- **positions**: array-like, default = [1, 2, ..., n]
  Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

- **vert**: bool, default = True.
  If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

- **widths**: array-like, default = 0.5
  Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

- **showmeans**: bool, default = False
If True, will toggle rendering of the means.

**showextrema**: bool, default = True
If True, will toggle rendering of the extrema.

**showmedians**: bool, default = False
If True, will toggle rendering of the medians.

**points**: scalar, default = 100
Defines the number of points to evaluate each of the gaussian kernel density estimations at.

**bw_method**: str, scalar or callable, optional
The method used to calculate the estimator bandwidth. This can be ‘scott’, ‘silverman’, a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a `GaussianKDE` instance as its only parameter and return a scalar. If None (default), ‘scott’ is used.

**Returns result**: dict
A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- **bodies**: A list of the `matplotlib.collections.PolyCollection` instances containing the filled area of each violin.
- **means**: A `matplotlib.collections.LineCollection` instance created to identify the mean values of each of the violin’s distribution.
- **mins**: A `matplotlib.collections.LineCollection` instance created to identify the bottom of each violin’s distribution.
- **maxes**: A `matplotlib.collections.LineCollection` instance created to identify the top of each violin’s distribution.
- **bars**: A `matplotlib.collections.LineCollection` instance created to identify the centers of each violin’s distribution.
- **medians**: A `matplotlib.collections.LineCollection` instance created to identify the median values of each of the violin’s distribution.

**vlines**(x, ymin, ymax, colors='k', linestyles='solid', label='', **kwargs)
Plot vertical lines.

Plot vertical lines at each x from ymin to ymax.

**Parameters**

- **x**: scalar or 1D array_like
  x-indexes where to plot the lines.
- **ymin, ymax**: scalar or 1D array_like
  Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- **colors**: array_like of colors, optional, default: ‘k’
- **linestyles**: ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional

949
**label**: string, optional, default: ''

**Returns lines**: `LineCollection`

**Other Parameters kwargs**: `LineCollection` properties.

See also:

**hlines** horizontal lines

**Examples**

![Vertical lines demo](image)

![Horizontal lines demo](image)

**xaxis_date**(tz=None)

Sets up x-axis ticks and labels that treat the x data as dates.

`tz` is a timezone string or tzinfo instance. Defaults to rc value.

**xaxis_inverted()**

Returns `True` if the x-axis is inverted.

**xcorr**(x, y, normed=True, detrend=<function detrend_none at 0x7f8bf872f320>, usevlines=True, maxlags=10, **kwargs)

Plot the cross correlation between `x` and `y`.

**Parameters**

- `x` : sequence of scalars of length `n`
- `y` : sequence of scalars of length `n`
- `hold` : boolean, optional, default: True
- `detrend` : callable, optional, default: `mlab.detrend_none`
  - `x` is detrended by the `detrend` callable. Default is no normalization.
- `normed` : boolean, optional, default: True
  - if True, normalize the data by the autocorrelation at the 0-th lag.
- `usevlines` : boolean, optional, default: True
  - if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.
- `maxlags` : integer, optional, default: 10
number of lags to show. If None, will return all $2 \times \text{len}(x) - 1$ lags.

**Returns** (lags, c, line, b) : where:
- lags are a length $2\times\text{maxlags}+1$ lag vector.
- c is the $2\times\text{maxlags}+1$ auto correlation vector.
- line is a **Line2D** instance returned by **plot**.
- b is the x-axis (none, if plot is used).

**Other Parameters**

**linestyle :** **Line2D** prop, optional, default: None
Only used if usevlines is False.

**marker :** string, optional, default: ‘o’

**Notes**

The cross correlation is performed with `numpy.correlate()` with mode = 2.

**yaxis_date**(tz=None)
Sets up y-axis ticks and labels that treat the y data as dates.

- `tz` is a timezone string or **tzinfo** instance. Defaults to rc value.

**yaxis_inverted**
Returns **True** if the y-axis is inverted.

**zorder = 0**
40.1 matplotlib.axis

Classes for the ticks and x and y axis

```python
class matplotlib.axis.Axis(axes, pickradius=15):
    Bases: matplotlib.artist.Artist

    Public attributes
    - axes.transData - transform data coords to display coords
    - axes.transAxes - transform axis coords to display coords
    - labelpad - number of points between the axis and its label

    Init the axis with the parent Axes instance

    OFFSETTEXTPAD = 3

    axis_date(tz=None)
    Sets up x-axis ticks and labels that treat the x data as dates. tz is a tzinfo instance or a timezone string. This timezone is used to create date labels.

    cla()
    clear the current axis

    convert_units(x)

    draw(artist, renderer, *args, **kwargs)
    Draw the axis lines, grid lines, tick lines and labels

    get_children()

    get_data_interval()
    return the Interval instance for this axis data limits

    get_gridlines()
    Return the grid lines as a list of Line2D instance

    get_label()
    Return the axis label as a Text instance
```
get_label_text()
    Get the text of the label

get_major_formatter()
    Get the formatter of the major ticker

get_major_locator()
    Get the locator of the major ticker

get_major_ticks(numticks=None)
    get the tick instances; grow as necessary

get_major_ticklabels()
    Return a list of Text instances for the major ticklabels

get_major_ticklines()
    Return the major tick lines as a list of Line2D instances

get_major_ticklocs()
    Get the major tick locations in data coordinates as a numpy array

get_minor_formatter()
    Get the formatter of the minor ticker

get_minor_locator()
    Get the locator of the minor ticker

get_minor_ticks(numticks=None)
    get the minor tick instances; grow as necessary

get_minor_ticklabels()
    Return a list of Text instances for the minor ticklabels

get_minor_ticklines()
    Return the minor tick lines as a list of Line2D instances

get_minor_ticklocs()
    Get the minor tick locations in data coordinates as a numpy array

get_offset_text()
    Return the axis offsetText as a Text instance

get_pickradius()
    Return the depth of the axis used by the picker

get_scale()

get_smart_bounds()
    get whether the axis has smart bounds

get_ticklabel_extents(renderer)
    Get the extents of the tick labels on either side of the axes.

get_ticklabels(minor=False, which=None)
    Get the x tick labels as a list of Text instances.
    Parameters
    minor : bool
If True return the minor ticklabels, else return the major ticklabels.

\textbf{which} : None, (‘minor’, ‘major’, ‘both’)

 Overrides \texttt{minor}.

Selects which ticklabels to return.

\textbf{Returns} \texttt{ret} : list

List of \texttt{Text} instances.

\texttt{get_ticklines}(\texttt{minor=False})

Return the tick lines as a list of Line2D instances.

\texttt{get_ticklocs}(\texttt{minor=False})

Get the tick locations in data coordinates as a numpy array.

\texttt{get_tightbbox}(\texttt{renderer})

Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and \texttt{offsetText}.

\texttt{get_transform}()

\texttt{get_units}()

return the units for axis.

\texttt{get_view_interval}()

return the Interval instance for this axis view limits.

\texttt{grid}(\texttt{b=None}, \texttt{which=’major’}, **\texttt{kwargs})

Set the axis grid on or off; \texttt{b} is a boolean. Use \texttt{which} = ‘major’ | ‘minor’ | ‘both’ to set the grid for major or minor ticks.

If \texttt{b} is \texttt{None} and len(\texttt{kwargs})==0, toggle the grid state. If \texttt{kwargs} are supplied, it is assumed you want the grid on and \texttt{b} will be set to True.

\texttt{kwargs} are used to set the line properties of the grids, e.g.,

\texttt{xax.grid(color=’r’, linestyle=’-’, linewidth=2)}

\texttt{have_units}()

\texttt{iter_ticks}()

Iterate through all of the major and minor ticks.

\texttt{limit_range_for_scale}(\texttt{vmin}, \texttt{vmax})

\texttt{pan}(\texttt{numsteps})

Pan \texttt{numsteps} (can be positive or negative)

\texttt{reset_ticks}()

\texttt{set_clip_path}(\texttt{clippath}, \texttt{transform=None})
set_data_interval()
    set the axis data limits

set_default_intervals()
    set the default limits for the axis data and view interval if they are not mutated

set_label_coords(x, y, transform=None)
    Set the coordinates of the label. By default, the x coordinate of the y label is determined by the
tick label bounding boxes, but this can lead to poor alignment of multiple ylabels if there are
multiple axes. Ditto for the y coordinate of the x label.
    You can also specify the coordinate system of the label with the transform. If None, the default
coordinate system will be the axes coordinate system (0,0) is (left,bottom), (0.5, 0.5) is middle,
etc

set_label_text(label, fontdict=None, **kwargs)
    Sets the text value of the axis label
    ACCEPTS: A string value for the label

set_major_formatter(formatter)
    Set the formatter of the major ticker
    ACCEPTS: A Formatter instance

set_major_locator(locator)
    Set the locator of the major ticker
    ACCEPTS: a Locator instance

set_minor_formatter(formatter)
    Set the formatter of the minor ticker
    ACCEPTS: A Formatter instance

set_minor_locator(locator)
    Set the locator of the minor ticker
    ACCEPTS: a Locator instance

set_pickradius(pickradius)
    Set the depth of the axis used by the picker
    ACCEPTS: a distance in points

set_scale(*args, **kwargs)
    Deprecated since version 1.3: The set_scale function was deprecated in version 1.3.
    This should be a private function (moved to _set_scale)

set_smart_bounds(value)
    set the axis to have smart bounds

set_tick_params(which='major', reset=False, **kw)
    Set appearance parameters for ticks and ticklabels.
    For documentation of keyword arguments, see matplotlib.axes.Axes.tick_params().
set_ticklabels(ticklabels, *args, **kwargs)
Set the text values of the tick labels. Return a list of Text instances. Use kwarg minor=True
to select minor ticks. All other kwargs are used to update the text object properties. As for
get_ticklabels, label1 (left or bottom) is affected for a given tick only if its label1On attribute is
True, and similarly for label2. The list of returned label text objects consists of all such label1
objects followed by all such label2 objects.

The input ticklabels is assumed to match the set of tick locations, regardless of the state of
label1On and label2On.

ACCEPTS: sequence of strings

set_ticks(ticks, minor=False)
Set the locations of the tick marks from sequence ticks

ACCEPTS: sequence of floats

set_units(u)
set the units for axis

ACCEPTS: a units tag

set_view_interval(vmin, vmax, ignore=False)

update_units(data)
introspect data for units converter and update the axis.converter instance if necessary. Return
True if data is registered for unit conversion.

zoom(direction)
Zoom in/out on axis; if direction is >0 zoom in, else zoom out

class matplotlib.axis.Tick(axes, loc, label, size=None, width=None, color=None, tick-
dir=None, pad=None, labelsize=None, labelcolor=None,
zorder=None, gridOn=None, tick1On=True, tick2On=True,
label1On=True, label2On=False, major=True)

Bases: matplotlib.artist.Artist

Abstract base class for the axis ticks, grid lines and labels

1 refers to the bottom of the plot for xticks and the left for yticks 2 refers to the top of the plot for
xticks and the right for yticks

Publicly accessible attributes:

tick1line a Line2D instance
tick2line a Line2D instance
gridline a Line2D instance
label1 a Text instance
label2 a Text instance
grdOn a boolean which determines whether to draw the tickline
tick1On a boolean which determines whether to draw the 1st tickline
tick2On a boolean which determines whether to draw the 2nd tickline
label1On a boolean which determines whether to draw tick label
label2On a boolean which determines whether to draw tick label
bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

**apply_tickdir**(tickdir)
Calculate self._pad and self._tickmarkers

**contains**(mouseevent)
Test whether the mouse event occurred in the Tick marks.
This function always returns false. It is more useful to test if the axis as a whole contains the mouse rather than the set of tick marks.

**draw**(artist, renderer, *args, **kwargs)

**get_children**()

**get_loc**()
Return the tick location (data coords) as a scalar

**get_pad**()
Get the value of the tick label pad in points

**get_pad_pixels**()

**get_view_interval**()
Return the view Interval instance for the axis this tick is ticking

**set_clip_path**(clippath, transform=None)
Set the artist’s clip path, which may be:
- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [(Path, Transform) | Patch | None]

**set_label**(s)
Set the text of ticklabel
ACCEPTS: str

**set_label1**(s)
Set the text of ticklabel
ACCEPTS: str

**set_label2**(s)
Set the text of ticklabel2
ACCEPTS: str
**set_pad(val)**
Set the tick label pad in points

ACCEPTS: float

class matplotlib.axis.Ticker

**formatter = None**

**locator = None**

class matplotlib.axis.XAxis(axes, pickradius=15)
Bases: matplotlib.axis.Axis

Init the axis with the parent Axes instance

**axis_name = u'x'**

**contains(mouseevent)**
Test whether the mouse event occurred in the x axis.

**get_data_interval()**
return the Interval instance for this axis data limits

**get_label_position()**
Return the label position (top or bottom)

**get_minpos()**

**get_text_heights(renderer)**
Returns the amount of space one should reserve for text above and below the axes. Returns a tuple (above, below)

**get_ticks_position()**
Return the ticks position (top, bottom, default or unknown)

**get_view_interval()**
return the Interval instance for this axis view limits

**set_data_interval(vmin, vmax, ignore=False)**
set the axis data limits

**set_default_intervals()**
set the default limits for the axis interval if they are not mutated

**set_label_position(position)**
Set the label position (top or bottom)

ACCEPTS: [‘top’ | ‘bottom’]

**set_ticks_position(position)**
Set the ticks position (top, bottom, both, default or none) both sets the ticks to appear on both
positions, but does not change the tick labels. ‘default’ resets the tick positions to the default:
ticks on both positions, labels at bottom. ‘none’ can be used if you don’t want any ticks. ‘none’
and ‘both’ affect only the ticks, not the labels.

ACCEPTS: [ ‘top’ | ‘bottom’ | ‘both’ | ‘default’ | ‘none’ ]

set_view_interval(vmin, vmax, ignore=False)
If ignore is False, the order of vmin, vmax does not matter; the original axis orientation will be
preserved. In addition, the view limits can be expanded, but will not be reduced. This method
is for mpl internal use; for normal use, see set_xlim().

tick_bottom()
use ticks only on bottom

tick_top()
use ticks only on top

class matplotlib.axis.XTick(axes, loc, label, size=None, width=None, color=None, tick-
dir=None, pad=None, labelsize=None, labelcolor=None, zorder=None, gridOn=None, tick1On=True, tick2On=True,
label1On=True, label2On=False, major=True)
Bases: matplotlib.axis.Tick
Contains all the Artists needed to make an x tick - the tick line, the label text and the grid line
bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords
size is the tick size in points

apply_tickdir(tickdir)

get_view_interval()
return the Interval instance for this axis view limits

update_position(loc)
Set the location of tick in data coords with scalar loc

class matplotlib.axis.YAxis(axes, pickradius=15)
Bases: matplotlib.axis.Axis
Init the axis with the parent Axes instance

axis_name = u’y’

contains(mouseevent)
Test whether the mouse event occurred in the y axis.

Returns True | False

get_data_interval()
return the Interval instance for this axis data limits

get_label_position()
Return the label position (left or right)
get_minpos()

get_text_widths(renderer)

get_ticks_position()
    Return the ticks position (left, right, both or unknown)

get_view_interval()
    return the Interval instance for this axis view limits

set_data_interval(vmin, vmax, ignore=False)
    set the axis data limits

set_default_intervals()
    set the default limits for the axis interval if they are not mutated

set_label_position(position)
    Set the label position (left or right)
    ACCEPTS: ['left' | 'right']

set_offset_position(position)

set_ticks_position(position)
    Set the ticks position (left, right, both, default or none) ‘both’ sets the ticks to appear on both positions, but does not change the tick labels. ‘default’ resets the tick positions to the default: ticks on both positions, labels at left. ‘none’ can be used if you don’t want any ticks. ‘none’ and ‘both’ affect only the ticks, not the labels.
    ACCEPTS: ['left' | 'right' | 'both' | 'default' | 'none']

set_view_interval(vmin, vmax, ignore=False)
    If ignore is False, the order of vmin, vmax does not matter; the original axis orientation will be preserved. In addition, the view limits can be expanded, but will not be reduced. This method is for mpl internal use; for normal use, see set ylim().

tick_left()
    use ticks only on left

tick_right()
    use ticks only on right

class matplotlib.axis.YTick(axes, loc, label, size=None, width=None, color=None,
tickdir=None, pad=None, labelsize=None, labelcolor=None,
zorder=None, gridOn=None, tick1On=True, tick2On=True,
label1On=True, label2On=False, major=True)
Bases: matplotlib.axis.Tick

Contains all the Artists needed to make a Y tick - the tick line, the label text and the grid line
bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points
apply_tickdir(tickdir)

get_view_interval()
    return the Interval instance for this axis view limits

update_position(loc)
    Set the location of tick in data coords with scalar loc
41.1 matplotlib.backend_bases

Abstract base classes define the primitives that renderers and graphics contexts must implement to serve as a matplotlib backend

**RendererBase**  An abstract base class to handle drawing/rendering operations.

**FigureCanvasBase**  The abstraction layer that separates the `matplotlib.figure.Figure` from the backend specific details like a user interface drawing area.

**GraphicsContextBase**  An abstract base class that provides color, line styles, etc...

**Event**  The base class for all of the matplotlib event handling. Derived classes such as `KeyEvent` and `MouseEvent` store the meta data like keys and buttons pressed, x and y locations in pixel and `Axes` coordinates.

**ShowBase**  The base class for the Show class of each interactive backend; the ‘show’ callable is then set to `Show.__call__`, inherited from ShowBase.

```python
class matplotlib.backend_bases.CloseEvent(name, canvas, guiEvent=None)
    Bases: matplotlib.backend_bases.Event

    An event triggered by a figure being closed

    In addition to the Event attributes, the following event attributes are defined:

class matplotlib.backend_bases.Cursors

    HAND = 0

    MOVE = 3

    POINTER = 1

    SELECT_REGION = 2
```
class matplotlib.backend_bases.DrawEvent
    name, canvas, renderer

Bases: matplotlib.backend_bases.Event

An event triggered by a draw operation on the canvas

In addition to the Event attributes, the following event attributes are defined:

- **renderer**: the RendererBase instance for the draw event

class matplotlib.backend_bases.Event
    name, canvas, guiEvent=None

A matplotlib event. Attach additional attributes as defined in FigureCanvasBase.mpl_connect().

The following attributes are defined and shown with their default values

- **name**: the event name
- **canvas**: the FigureCanvas instance generating the event
- **guiEvent**: the GUI event that triggered the matplotlib event

class matplotlib.backend_bases.FigureCanvasBase
    figure

Bases: object

The canvas the figure renders into.

Public attributes

- **figure**: A matplotlib.figure.Figure instance

blit
    bbox=None

blit the canvas in bbox (default entire canvas)

button_press_event
    x, y, button, dblclick=False, guiEvent=None

Backend derived classes should call this function on any mouse button press. x,y are the canvas
coords: 0,0 is lower, left. button and key are as defined in MouseEvent.

This method will be call all functions connected to the ‘button_press_event’ with a MouseEvent
instance.

button_release_event
    x, y, button, guiEvent=None

Backend derived classes should call this function on any mouse button release.

x the canvas coordinates where 0=left
y the canvas coordinates where 0=bottom

guiEvent the native UI event that generated the mpl event

This method will be call all functions connected to the ‘button_release_event’ with a
MouseEvent instance.

close_event
    guiEvent=None

This method will be called by all functions connected to the ‘close_event’ with a CloseEvent

draw(*args, **kwargs)

Render the Figure

draw_cursor
    event

Draw a cursor in the event.axes if inaxes is not None. Use native GUI drawing for efficiency if
possible

draw_event
    renderer

This method will be call all functions connected to the ‘draw_event’ with a DrawEvent

draw_idle(*args, **kwargs)

draw() only if idle; defaults to draw but backends can override
Matplotlib, Release 1.4.3

enter_notify_event(guiEvent=None, xy=None)
Backend derived classes should call this function when entering canvas

guiEvent the native UI event that generated the mpl event
xy the coordinate location of the pointer when the canvas is entered

events = ['resize_event', 'draw_event', 'key_press_event', 'key_release_event', 'button_press_event', 'button_release_event',
          'scroll_event', 'motion_notify_event', 'motion_leave_event', 'figure_enter_event', 'figure_leave_event', 'axes_enter_event',
          'axes_leave_event', 'cursor_event', 'button_press_event', 'button_release_event', 'key_press_event',
          'key_release_event', 'scroll_event', 'motion_notify_event', 'motion_leave_event', 'figure_enter_event',
          'figure_leave_event', 'axes_enter_event', 'axes_leave_event', 'cursor_event', 'button_press_event',
          'button_release_event', 'key_press_event', 'key_release_event', 'scroll_event', 'motion_notify_event',
          'motion_leave_event', 'figure_enter_event', 'figure_leave_event', 'axes_enter_event', 'axes_leave_event',
          'cursor_event', 'button_press_event', 'button_release_event', 'key_press_event', 'key_release_event',
          'scroll_event', 'motion_notify_event', 'motion_leave_event', 'figure_enter_event',
          'figure_leave_event', 'axes_enter_event', 'axes_leave_event', 'cursor_event', 'button_press_event',
          'button_release_event', 'key_press_event', 'key_release_event', 'scroll_event', 'motion_notify_event',
          'motion_leave_event', 'figure_enter_event', 'figure_leave_event', 'axes_enter_event',
          'axes_leave_event', 'cursor_event', 'button_press_event', 'button_release_event', 'key_press_event',
          'key_release_event', 'scroll_event', 'motion_notify_event', 'motion_leave_event', 'figure_enter_event',
          'figure_leave_event', 'axes_enter_event', 'axes_leave_event', 'cursor_event', 'button_press_event',
          'button_release_event', 'key_press_event', 'key_release_event', 'scroll_event', 'motion_notify_event',
          'motion_leave_event', 'figure_enter_event', 'figure_leave_event', 'axes_enter_event', 'axes_leave_event',
          'cursor_event', 'button_press_event', 'button_release_event', 'key_press_event', 'key_release_event',
          'scroll_event', 'motion_notify_event', 'motion_leave_event', 'figure_enter_event', 'figure_leave_event',
          'axes_enter_event', 'axes_leave_event', 'cursor_event', 'button_press_event', 'button_release_event',
          'key_press_event', 'key_release_event', 'scroll_event', 'motion_notify_event', 'motion_leave_event',
          'figure_enter_event', 'figure_leave_event', 'axes_enter_event', 'axes_leave_event',
          'cursor_event', 'button_press_event', 'button_release_event', 'key_press_event', 'key_release_event',
          'scroll_event', 'motion_notify_event', 'motion_leave_event', 'figure_enter_event', 'figure_leave_event',
          'axes_enter_event', 'axes_leave_event', 'cursor_event', 'button_press_event', 'button_release_event',
          'key_press_event', 'key_release_event', 'scroll_event', 'motion_notify_event', 'motion_leave_event',
          'figure_enter_event', 'figure_leave_event', 'axes_enter_event', 'axes_leave_event',
          'cursor_event', 'button_press_event', 'button_release_event', 'key_press_event', 'key_release_event',
          'scroll_event', 'motion_notify_event', 'motion_leave_event',

filetypes = {'pgf': 'PGF code for LaTeX', 'svgz': 'Scalable Vector Graphics', 'tiff': 'Tagged Image File Format',
             'jpg': 'Joint Photographic Experts Group', 'raw': 'Raw RGBA bitmap', 'jpeg': 'Joint Photographic Experts Group',
             'pdf': 'Portable Document Format', 'tif': 'Tagged Image File Format'}

fixed_dpi = None

flush_events()
Flush the GUI events for the figure. Implemented only for backends with GUIs.

get_default_filename()
Return a string, which includes extension, suitable for use as a default filename.

classmethod get_default_filetype()
Get the default savefig file format as specified in rcParam savefig.format. Returned string
excludes period. Overridden in backends that only support a single file type.

classmethod get_supported_filetypes()
Return dict of savefig file formats supported by this backend

classmethod get_supported_filetypes_grouped()
Return a dict of savefig file formats supported by this backend, where the keys are a file type
name, such as ‘Joint Photographic Experts Group’, and the values are a list of filename
extensions used for that filetype, such as [‘jpg’, ‘jpeg’].

get_width_height()
Return the figure width and height in points or pixels (depending on the backend), truncated to
integers

get_window_title()
Get the title text of the window containing the figure. Return None if there is no window (e.g.,
a PS backend).

grab_mouse(ax)
Set the child axes which are currently grabbing the mouse events. Usually called by the widgets
themselves. It is an error to call this if the mouse is already grabbed by another axes.

idle_event(guiEvent=None)
Called when GUI is idle.

is_saving()
Returns True when the renderer is in the process of saving to a file, rather than rendering for
an on-screen buffer.

key_press_event(key, guiEvent=None)
This method will be call all functions connected to the ‘key_press_event’ with a KeyEvent
key_release_event(key, guiEvent=None)
This method will be call all functions connected to the ‘key_release_event’ with a KeyEvent

leave_notify_event(guiEvent=None)
Backend derived classes should call this function when leaving canvas

guiEvent the native UI event that generated the mpl event

motion_notify_event(x, y, guiEvent=None)
Backend derived classes should call this function on any motion-notify-event.
x the canvas coordinates where 0=left
y the canvas coordinates where 0=bottom

guiEvent the native UI event that generated the mpl event
This method will be call all functions connected to the ‘motion_notify_event’ with a MouseEvent instance.

mpl_connect(s, func)
Connect event with string s to func. The signature of func is:

```python
def func(event)
```

where event is a matplotlib.backend_bases.Event. The following events are recognized
• 'button_press_event'
• 'button_release_event'
• 'draw_event'
• 'key_press_event'
• 'key_release_event'
• 'motion_notify_event'
• 'pick_event'
• 'resize_event'
• 'scroll_event'
• 'figure_enter_event',
• 'figure_leave_event',
• 'axes_enter_event',
• 'axes_leave_event'
• 'close_event'

For the location events (button and key press/release), if the mouse is over the axes, the variable

event.inaxes will be set to the Axes the event occurs is over, and additionally, the variables

event.xdata and event.ydata will be defined. This is the mouse location in data coords. See KeyEvent and MouseEvent for more info.

Return value is a connection id that can be used with mpl_disconnect().

Example usage:

```python
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

mpl_disconnect(cid)
Disconnected callback id cid
Example usage:

cid = canvas.mpl_connect('button_press_event', on_press)
#...later
canvas.mpl_disconnect(cid)

**new_timer**(\*\*args, \*\*kwargs)**

Creates a new backend-specific subclass of backend_bases.Timer. This is useful for getting periodic events through the backend’s native event loop. Implemented only for backends with GUIs.

optional arguments:

*interval* Timer interval in milliseconds
*callbacks* Sequence of (func, args, kwargs) where func(args, **kwargs) will be executed by the timer every *interval.*

**onHilite**(ev)**

Mouse event processor which highlights the artists under the cursor. Connect this to the `motion_notify_event` using:

canvas.mpl_connect('motion_notify_event', canvas.onHilite)

**onRemove**(ev)**

Mouse event processor which removes the top artist under the cursor. Connect this to the `mouse_press_event` using:

canvas.mpl_connect('mouse_press_event', canvas.onRemove)

**pick**(mouseevent)**

**pick_event**(mouseevent, artist, **kwargs)**

This method will be called by artists who are picked and will fire off PickEvent callbacks registered listeners

**print_figure**(filename, dpi=None, facecolor='w', edgecolor='w', orientation='portrait', format=\*None, **kwargs)**

Render the figure to hardcopy. Set the figure patch face and edge colors. This is useful because some of the GUIs have a gray figure face color background and you’ll probably want to override this on hardcopy.

Arguments are:

*filename* can also be a file object on image backends
*orientation* only currently applies to PostScript printing.
*dpi* the dots per inch to save the figure in; if None, use savefig.dpi
*facecolor* the facecolor of the figure
*edgecolor* the edgecolor of the figure
*orientation* landscape’ | ‘portrait’ (not supported on all backends)
*format* when set, forcibly set the file format to save to
*bbox_inches* Bbox in inches. Only the given portion of the figure is saved. If ‘tight’, try to figure out the tight bbox of the figure. If None, use savefig.bbox
pad_inches Amount of padding around the figure when bbox_inches is ‘tight’. If None, use savefig.pad_inches
bbox_extra_artists A list of extra artists that will be considered when the tight bbox is calculated.

release_mouse(ax)
Release the mouse grab held by the axes, ax. Usually called by the widgets. It is ok to call this even if you ax doesn’t have the mouse grab currently.

resize(w, h)
set the canvas size in pixels

resize_event() This method will be call all functions connected to the ‘resize_event’ with a ResizeEvent

scroll_event(x, y, step, guiEvent=None)
Backend derived classes should call this function on any scroll wheel event. x,y are the canvas coords: 0,0 is lower, left. button and key are as defined in MouseEvent.
This method will be call all functions connected to the ‘scroll_event’ with a MouseEvent instance.

set_window_title(title)
Set the title text of the window containing the figure. Note that this has no effect if there is no window (e.g., a PS backend).

start_event_loop(timeout)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.
This is implemented only for backends with GUIs.

start_event_loop_default(timeout=0)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.
This function provides default event loop functionality based on time.sleep that is meant to be used until event loop functions for each of the GUI backends can be written. As such, it throws a deprecated warning.

Call signature:

    start_event_loop_default(self, timeout=0)

This call blocks until a callback function triggers stop_event_loop() or timeout is reached. If timeout is <=0, never timeout.

stop_event_loop() Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events.
This is implemented only for backends with GUIs.
stop_event_loop_default()
Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events.

Call signature:

stop_event_loop_default(self)

supports_blit = True

switch_backends(FigureCanvasClass)
Instantiate an instance of FigureCanvasClass
This is used for backend switching, e.g., to instantiate a FigureCanvasPS from a FigureCanvasGTK. Note, deep copying is not done, so any changes to one of the instances (e.g., setting figure size or line props), will be reflected in the other

class matplotlib.backend_bases.FigureManagerBase(canvas, num)
Bases: object
Helper class for pyplot mode, wraps everything up into a neat bundle
Public attributes:
canvas A FigureCanvasBase instance
num The figure number
destroy()

full_screen_toggle()

get_window_title()
Get the title text of the window containing the figure. Return None for non-GUI backends (e.g., a PS backend).

key_press(event)
Implement the default mpl key bindings defined at Navigation Keyboard Shortcuts

key_press_handler_id = None
The returned id from connecting the default key handler via FigureCanvasBase.mpl_connect().
To disable default key press handling:

manager, canvas = figure.canvas.manager, figure.canvas
canvas.mpl_disconnect(manager.key_press_handler_id)

resize(w, h)
“For gui backends, resize the window (in pixels).

set_window_title(title)
Set the title text of the window containing the figure. Note that this has no effect for non-GUI backends (e.g., a PS backend).
**show**
For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception to be caught by `show()`, for an optional warning.

**show_popup** *(msg)*
Display message in a popup – GUI only

```python
class matplotlib.backend_bases.GraphicsContextBase
An abstract base class that provides color, line styles, etc...

copy_properties(gc)
Copy properties from gc to self

dashd = {u'solid': (None, None), u'dashed': (0, (6.0, 6.0)), u'dotted': (0, (1.0, 3.0)), u'dashdot': (0, (3.0, 5.0, 1.0, 5.0))}

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_antialiased()
Return true if the object should try to do antialiased rendering

get_capstyle()
Return the capstyle as a string in (‘butt’, ‘round’, ‘projecting’)

get_clip_path()
Return the clip path in the form (path, transform), where path is a Path instance, and transform is an affine transform to apply to the path before clipping.

get_clip_rectangle()
Return the clip rectangle as a Bbox instance

get_dashes()
Return the dash information as an offset dashlist tuple.

The dash list is a even size list that gives the ink on, ink off in pixels.

See p107 of to PostScript BLUEBOOK for more info.

Default value is None

get_forced_alpha()
Return whether the value given by get_alpha() should be used to override any other alpha-channel values.

get_gid()
Return the object identifier if one is set, None otherwise.

get_hatch()
Gets the current hatch style

get_hatch_path(*density=6.0*)
Returns a Path for the current hatch.

get_joinstyle()
Return the line join style as one of (‘miter’, ‘round’, ‘bevel’)
```
get_linestyle(style)
Return the linestyle: one of ('solid', 'dashed', 'dashdot', 'dotted').

get_linewidth()
Return the line width in points as a scalar

geti_rgbi()
returns a tuple of three or four floats from 0-1.

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None
A 3-tuple with the following elements:
  • scale: The amplitude of the wiggle perpendicular to the source line.
  • length: The length of the wiggle along the line.
  • randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

get_snap()
returns the snap setting which may be:
  • True: snap vertices to the nearest pixel center
  • False: leave vertices as-is
  • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

get_url()
returns a url if one is set, None otherwise

restore()
Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack

set_alpha(alpha)
Set the alpha value used for blending - not supported on all backends. If alpha=None (the default), the alpha components of the foreground and fill colors will be used to set their respective transparencies (where applicable); otherwise, alpha will override them.

set_anti.aliased(b)
True if object should be drawn with antialiased rendering

set_capstyle(cs)
Set the capstyle as a string in ('butt', 'round', 'projecting')

set_clip_path(path)
Set the clip path and transformation. Path should be a TransformedPath instance.

set_clip_rectangle(rectangle)
Set the clip rectangle with sequence (left, bottom, width, height)

set_dashes(dash_offset, dash_list)
Set the dash style for the gc.

 dash_offset is the offset (usually 0).
`dash_list` specifies the on-off sequence as points. `(None, None)` specifies a solid line.

**set_foreground**(fg, isRGBA=False)
Set the foreground color. fg can be a MATLAB format string, a html hex color string, an rgb or rgba unit tuple, or a float between 0 and 1. In the latter case, grayscale is used.

If you know fg is rgba, set isRGBA=True for efficiency.

**set_gid**(id)
Sets the id.

**set_graylevel**(frac)
Set the foreground color to be a gray level with frac.

**set_hatch**(hatch)
Sets the hatch style for filling.

**set_joinstyle**(js)
Set the join style to be one of (‘miter’, ‘round’, ‘bevel’)

**set_linestyle**(style)
Set the linestyle to be one of (‘solid’, ‘dashed’, ‘dashdot’, ‘dotted’). One may specify customized dash styles by providing a tuple of (offset, dash pairs). For example, the predefined line styles have following values:

- ‘dashed’ : (0, (6.0, 6.0)), ‘dashdot’ : (0, (3.0, 5.0, 1.0, 5.0)), ‘dotted’ : (0, (1.0, 3.0)),

**set_linewidth**(w)
Set the linewidth in points.

**set_sketch_params**(scale=None, length=None, randomness=None)
Sets the the sketch parameters.

**Parameters**
- `scale` : float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.
- `length` : float, optional
  The length of the wiggle along the line, in pixels (default 128.0)
- `randomness` : float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

**set_url**(url)
Sets the url for links in compatible backends.

**class** matplotlib.backend_bases.IdleEvent(name, canvas, guiEvent=None)
Bases: matplotlib.backend_bases.Event
An event triggered by the GUI backend when it is idle – useful for passive animation

```python
class matplotlib.backend_bases.KeyEvent(name, canvas, key, x=0, y=0, guiEvent=None)
Bases: matplotlib.backend_bases.LocationEvent

A key event (key press, key release).

Attach additional attributes as defined in FigureCanvasBase.mpl_connect().

In addition to the Event and LocationEvent attributes, the following attributes are defined:

- **key** the key(s) pressed. Could be **None**, a single case sensitive ascii character ("g", "G", ",", etc.), a special key ("control", "shift", "f1", "up", etc.) or a combination of the above (e.g., 
  "ctrl+alt+g", "ctrl+alt+G").

**Note:** Modifier keys will be prefixed to the pressed key and will be in the order "ctrl", "alt", "super". The exception to this rule is when the pressed key is itself a modifier key, therefore "ctrl+alt" and "alt+control" can both be valid key values.
```

Example usage:

```python
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

```python
class matplotlib.backend_bases.LocationEvent(name, canvas, x, y, guiEvent=None)
Bases: matplotlib.backend_bases.Event

An event that has a screen location

The following additional attributes are defined and shown with their default values.

In addition to the Event attributes, the following event attributes are defined:

- **x** x position - pixels from left of canvas
- **y** y position - pixels from bottom of canvas
- **inaxes** the Axes instance if mouse is over axes
- **xdata** x coord of mouse in data coords
- **ydata** y coord of mouse in data coords
- **x**, **y** in figure coords, 0,0 = bottom, left

inaxes = None

date=_lastevent = None

x = None

xdata = None

y = None
```
ydata = None

class matplotlib.backend_bases.MouseEvent (name, canvas, x, y, button=None, key=None, step=0, dblclick=False, guiEvent=None)

Bases: matplotlib.backend_bases.LocationEvent

A mouse event (‘button_press_event’, ‘button_release_event’, ‘scroll_event’, ‘motion_notify_event’).

In addition to the Event and LocationEvent attributes, the following attributes are defined:

**button** button pressed None, 1, 2, 3, ‘up’, ‘down’ (up and down are used for scroll events)

**key** the key depressed when the mouse event triggered (see KeyEvent)

**step** number of scroll steps (positive for ‘up’, negative for ‘down’)

Example usage:

def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('button_press_event', on_press)

x, y in figure coords, 0,0 = bottom, left

button = None

dblclick = None

inaxes = None

step = None

x = None

xdata = None

y = None

ydata = None

class matplotlib.backend_bases.NavigationToolbar2(canvas)

Bases: object

Base class for the navigation cursor, version 2

backends must implement a canvas that handles connections for ‘button_press_event’ and ‘button_release_event’. See FigureCanvasBase.mpl_connect() for more information

They must also define

save_figure() save the current figure
set_cursor() if you want the pointer icon to change
_init_toolbar() create your toolbar widget
draw_rubberband() (optional) draw the zoom to rect “rubberband” rectangle
press() (optional) whenever a mouse button is pressed, you’ll be notified with the event
release() (optional) whenever a mouse button is released, you’ll be notified with the event
dynamic_update() (optional) dynamically update the window while navigating
set_message() (optional) display message
set_history_buttons() (optional) you can change the history back / forward buttons
to indicate disabled / enabled state.

That’s it, we’ll do the rest!

back(*args)
move back up the view lim stack
drag_pan(event)
the drag callback in pan/zoom mode
drag_zoom(event)
the drag callback in zoom mode
draw()
Redraw the canvases, update the locators
draw_rubberband(event, x0, y0, x1, y1)
Draw a rectangle rubberband to indicate zoom limits
dynamic_update()

forward(*args)
Move forward in the view lim stack
home(*args)
Restore the original view
mouse_move(event)

pan(*args)
Activate the pan/zoom tool. pan with left button, zoom with right
press(event)
Called whenever a mouse button is pressed.
press_pan(event)
the press mouse button in pan/zoom mode callback
press_zoom(event)
the press mouse button in zoom to rect mode callback
push_current()
push the current view limits and position onto the stack
release(event)
    this will be called whenever mouse button is released

release_pan(event)
    the release mouse button callback in pan/zoom mode

release_zoom(event)
    the release mouse button callback in zoom to rect mode

save_figure(*args)
    Save the current figure

set_cursor(cursor)
    Set the current cursor to one of the Cursors enums values

set_history_buttons()
    Enable or disable back/forward button

set_message(s)
    Display a message on toolbar or in status bar

toolitems = ((u'Home', u'Reset original view', u'home', u'home'), (u'Back', u'Back to previous view', u'back', u'back'), (u'Forward', u'Forward to next view', u'forward', u'forward'), (u'Configure subplots', u'subplots', u'configure_subplots'), (u'Save', u'Save the figure', u'filesave', u'save_figure'))

update()
    Reset the axes stack

zoom(*args)
    Activate zoom to rect mode

def exception matplotlib.backend_bases.NonGuiException
Bases: exceptions.Exception

class matplotlib.backend_bases.PickEvent(name, canvas, mouseevent, artist, guiEvent=None, **kwargs)
Bases: matplotlib.backend_bases.Event

A pick event, fired when the user picks a location on the canvas sufficiently close to an artist.

Attrs: all the Event attributes plus

mouseevent the MouseEvent that generated the pick

artist the Artist picked

other extra class dependent attrs – e.g., a Line2D pick may define different extra attributes than a PatchCollection pick event

Example usage:

line, = ax.plot(rand(100), 'o', picker=5) # 5 points tolerance

def on_pick(event):
    thisline = event.artist
    xdata, ydata = thisline.get_data()
    ind = event.ind
    print('on pick line:', zip(xdata[ind], ydata[ind]))

    cid = fig.canvas.mpl_connect('pick_event', on_pick)
class matplotlib.backend_bases.RendererBase

Bases: object

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implement-
ing draw_path() alone would give a highly capable backend):

- draw_path()
- draw_image()
- draw_gouraud_triangle()

The following methods should be implemented in the backend for optimization reasons:

- draw_text()
- draw_markers()
- draw_path_collection()
- draw_quad_mesh()

close_group(s)

Close a grouping element with label s Is only currently used by backend_svg

draw_gouraud_triangle(gc, points, colors, transform)

Draw a Gouraud-shaded triangle.

points is a 3x2 array of (x, y) points for the triangle.

colors is a 3x4 array of RGBA colors for each point of the triangle.

transform is an affine transform to apply to the points.

draw_gouraud_triangles(gc, triangles_array, colors_array, transform)

Draws a series of Gouraud triangles.

points is a Nx3x2 array of (x, y) points for the triangle.

colors is a Nx3x4 array of RGBA colors for each point of the triangles.

transform is an affine transform to apply to the points.

draw_image(gc, x, y, im)

Draw the image instance into the current axes;

gc a GraphicsContext containing clipping information

x is the distance in pixels from the left hand side of the canvas.

y the distance from the origin. That is, if origin is upper, y is the distance from top. If origin is

lower, y is the distance from bottom

im the matplotlib._image.Image instance

draw_markers(gc, marker_path, marker_trans, path, trans, rgbFace=None)

Draws a marker at each of the vertices in path. This includes all vertices, including control

points on curves. To avoid that behavior, those vertices should be removed before calling this

function.

gc the GraphicsContextBase instance

marker_trans is an affine transform applied to the marker.

trans is an affine transform applied to the path.

This provides a fallback implementation of draw_markers that makes multiple calls to
draw_path(). Some backends may want to override this method in order to draw the marker

only once and reuse it multiple times.
draw_path(gc, path, transform, rgbFace=None)
Draws a Path instance using the given affine transform.

draw_path_collection(gc, master_transform, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds, urls, offset_position)
Draws a collection of paths selecting drawing properties from the lists facecolors, edgecolors, linewidths, linestyles and antialiaseds. offsets is a list of offsets to apply to each of the paths. The offsets in offsets are first transformed by offsetTrans before being applied. offset_position may be either “screen” or “data” depending on the space that the offsets are in.

This provides a fallback implementation of draw_path_collection() that makes multiple calls to draw_path(). Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods _iter_collection_raw_paths() and _iter_collection() are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of draw_path_collection() can be made globally.

draw_quad_mesh(gc, master_transform, meshWidth, meshHeight, coordinates, offsets, offsetTrans, facecolors, antialiased, edgecolors)
This provides a fallback implementation of draw_quad_mesh() that generates paths and then calls draw_path_collection().

draw_tex(gc, x, y, s, prop, angle, ismath=u'TeX!', mtext=None)

draw_text(gc, x, y, s, prop, angle, ismath=False, mtext=None)
Draw the text instance

gc the GraphicsContextBase instance
x the x location of the text in display coords
y the y location of the text baseline in display coords
s the text string
prop a matplotlib.font_manager.FontProperties instance
angle the rotation angle in degrees
mtext a matplotlib.text.Text instance

backend implementers note

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

if 0: bbox_artist(self, renderer)

to if 1, and then the actual bounding box will be plotted along with your text.

flipy()
Return true if y small numbers are top for renderer Is used for drawing text (matplotlib.text) and images (matplotlib.image) only

get_canvas_width_height()
return the canvas width and height in display coords
get_image_magnification()
Get the factor by which to magnify images passed to draw_image(). Allows a backend to have images at a different resolution to other artists.

get_texmanager()
return the matplotlib.texmanager.TexManager instance

get_text_width_height_descent(s, prop, ismath)
get the width and height, and the offset from the bottom to the baseline (descent), in display coords of the string s with FontProperties prop

new_gc()
Return an instance of a GraphicsContextBase

open_group(s, gid=None)
Open a grouping element with label s. If gid is given, use gid as the id of the group. Is only currently used by backend_svg.

option_image_nocomposite()
override this method for renderers that do not necessarily want to rescale and composite raster images. (like SVG)

option_scale_image()
override this method for renderers that support arbitrary scaling of image (most of the vector backend).

points_to_pixels(points)
Convert points to display units
points a float or a numpy array of float
return points converted to pixels

You need to override this function (unless your backend doesn’t have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

points to pixels = points * pixels_per_inch/72.0 * dpi/72.0

start_filter()
Used in AggRenderer. Switch to a temporary renderer for image filtering effects.

start_rasterizing()
Used in MixedModeRenderer. Switch to the raster renderer.

stop_filter(filter_func)
Used in AggRenderer. Switch back to the original renderer. The contents of the temporary renderer is processed with the filter_func and is drawn on the original renderer as an image.

stop_rasterizing()
Used in MixedModeRenderer. Switch back to the vector renderer and draw the contents of the raster renderer as an image on the vector renderer.

strip_math(s)
class matplotlib.backend_bases.ResizeEvent(name, canvas)

Bases: matplotlib.backend_bases.Event

An event triggered by a canvas resize

In addition to the Event attributes, the following event attributes are defined:

- **width**: width of the canvas in pixels
- **height**: height of the canvas in pixels

class matplotlib.backend_bases.ShowBase

Bases: object

Simple base class to generate a show() callable in backends.

Subclass must override mainloop() method.

```python
mainloop()
```

class matplotlib.backend_bases.TimerBase(interval=None, callbacks=None)

Bases: object

A base class for providing timer events, useful for things animations. Backends need to implement a few specific methods in order to use their own timing mechanisms so that the timer events are integrated into their event loops.

Mandatory functions that must be implemented:

- **_timer_start**: Contains backend-specific code for starting the timer
- **_timer_stop**: Contains backend-specific code for stopping the timer

Optional overrides:

- **_timer_set_single_shot**: Code for setting the timer to single shot operating mode, if supported by the timer object. If not, the Timer class itself will store the flag and the _on_timer method should be overridden to support such behavior.
- **_timer_set_interval**: Code for setting the interval on the timer, if there is a method for doing so on the timer object.
- **_on_timer**: This is the internal function that any timer object should call, which will handle the task of running all callbacks that have been set.

Attributes:

- **interval**: The time between timer events in milliseconds. Default is 1000 ms.
- **single_shot**: Boolean flag indicating whether this timer should operate as single shot (run once and then stop). Defaults to False.
- **callbacks**: Stores list of (func, args) tuples that will be called upon timer events. This list can be manipulated directly, or the functions add_callback and remove_callback can be used.

```python
add_callback(func, *args, **kwargs)
```

Register func to be called by timer when the event fires. Any additional arguments provided will be passed to func.

```python
interval
```

```python
remove_callback(func, *args, **kwargs)
```

Remove func from list of callbacks. args and kwargs are optional and used to distinguish between copies of the same function registered to be called with different arguments.
single_shot

**start**(interval=None)

Start the timer object. interval is optional and will be used to reset the timer interval first if provided.

**stop()**

Stop the timer.

matplotlib.backend_bases.get_registered_canvas_class(format)

Return the registered default canvas for given file format. Handles deferred import of required backend.

matplotlib.backend_bases.key_press_handler(event, canvas, toolbar=None)

Implement the default mpl key bindings for the canvas and toolbar described at Navigation Keyboard Shortcuts.

- event a KeyEvent instance
- canvas a FigureCanvasBase instance
- toolbar a NavigationToolbar2 instance

matplotlib.backend_bases.register_backend(format, backend, description=None)

Register a backend for saving to a given file format.

- format [str] File extension
- backend [module string or canvas class] Backend for handling file output
- description [str, optional] Description of the file type. Defaults to an empty string

### 41.2 matplotlib.backends.backend_gtkagg

TODO We'll add this later, importing the gtk backends requires an active X-session, which is not compatible with cron jobs.

### 41.3 matplotlib.backends.backend_qt4agg

Render to qt from agg

matplotlib.backends.backend_qt4agg.FigureCanvas

alias of FigureCanvasQTAgg

class matplotlib.backends.backend_qt4agg.FigureCanvasQTAgg(fig)

Bases: matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase, matplotlib.backends.backend_qt4.FigureCanvasQT, matplotlib.backends.backend_agg.FigureCanvas

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

Public attribute

- figure - A Figure instance

matplotlib.backends.backend_qt4agg.new_figure_manager(num, *args, **kwargs)

Create a new figure manager instance


41.4 matplotlib.backends.backend_wxagg

matplotlib.backends.backend_wxagg.FigureCanvas
alias of FigureCanvasWxAgg
class matplotlib.backends.backend_wxagg.FigureCanvasWxAgg(parent, id, figure)
    Bases: matplotlib.backends.backend_agg.FigureCanvasAgg, matplotlib.backends.backend_wx.FigureCanvasWx
    The FigureCanvas contains the figure and does event handling.
    In the wxPython backend, it is derived from wxPanel, and (usually) lives inside a frame instantiated by a FigureManagerWx. The parent window probably implements a wxSizer to control the displayed control size - but we give a hint as to our preferred minimum size.
    Initialise a FigureWx instance.
        • Initialise the FigureCanvasBase and wxPanel parents.
        • Set event handlers for: EVT_SIZE (Resize event) EVT_PAINT (Paint event)
    blit(bbox=None)
        Transfer the region of the agg buffer defined by bbox to the display. If bbox is None, the entire buffer is transferred.
    draw(drawDC=None)
        Render the figure using agg.
    print_figure(filename, *args, **kwargs)

class matplotlib.backends.backend_wxagg.FigureFrameWxAgg(num, fig)
    Bases: matplotlib.backends.backend_wx.FigureFrameWx
    get_canvas(fig)

class matplotlib.backends.backend_wxagg.NavigationToolbar2WxAgg(canvas)
    Bases: matplotlib.backends.backend_wx.NavigationToolbar2Wx
    get_canvas(frame, fig)

matplotlib.backends.backend_wxagg.new_figure_manager(num, *args, **kwargs)
    Create a new figure manager instance
matplotlib.backends.backend_wxagg.new_figure_manager_given_figure(num, figure)
    Create a new figure manager instance for the given figure.
41.5 matplotlib.backends.backend_pdf

A PDF matplotlib backend Author: Jouni K Seppälä <jks@iki.fi>

matplotlib.backends.backend_pdf.FigureCanvas
alias of FigureCanvasPdf

class matplotlib.backends.backend_pdf.FigureCanvasPdf(figure)
   Bases: matplotlib.backend_bases.FigureCanvasBase
   The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...
   
   Public attribute
   figure - A Figure instance

class matplotlib.backends.backend_pdf.Name(name)
   Bases: object
   PDF name object.

class matplotlib.backends.backend_pdf.Operator(op)
   Bases: object
   PDF operator object.

class matplotlib.backends.backend_pdf.PdfFile(filename)
   Bases: object
   PDF file object.

   alphaState(alpha)
       Return name of an ExtGState that sets alpha to the given value

   embedTTF(filename, characters)
       Embed the TTF font from the named file into the document.

   fontName(fontprop)
       Select a font based on fontprop and return a name suitable for Op.selectfont. If fontprop is a string, it will be interpreted as the filename (or dvi name) of the font.

   imageObject(image)
       Return name of an image XObject representing the given image.

   markerObject(path, trans, fillp, strokep, lw, joinstyle, capstyle)
       Return name of a marker XObject representing the given path.

   reserveObject(name='u')
       Reserve an ID for an indirect object. The name is used for debugging in case we forget to print out the object with writeObject.

   writeInfoDict()
       Write out the info dictionary, checking it for good form

   writeTrailer()
       Write out the PDF trailer.
writeXref()
Write out the xref table.

class matplotlib.backends.backend_pdf.PdfPages(filename, keep_empty=True)
Bases: object
A multi-page PDF file.

Notes

In reality PdfPages is a thin wrapper around PdfFile, in order to avoid confusion when using savefig() and forgetting the format argument.

Examples

```python
>>> import matplotlib.pyplot as plt
>>> # Initialize:
>>> with PdfPages('foo.pdf') as pdf:
...     # As many times as you like, create a figure fig and save it:
...     fig = plt.figure()
...     pdf.savefig(fig)
...     # When no figure is specified the current figure is saved
...     pdf.savefig()
```

Create a new PdfPages object.

Parameters filename: str
Plots using PdfPages.savefig() will be written to a file at this location. The file is opened at once and any older file with the same name is overwritten.

keep_empty: bool, optional
If set to False, then empty pdf files will be deleted automatically when closed.

close()
Finalize this object, making the underlying file a complete PDF file.

get_pagecount()
Returns the current number of pages in the multipage pdf file.

infodict()
Return a modifiable information dictionary object (see PDF reference section 10.2.1 ‘Document Information Dictionary’).

savefig(figure=None, **kwargs)
Saves a Figure to this file as a new page.

Any other keyword arguments are passed to savefig().

Parameters figure: :class:`matplotlib.figure.Figure` or int, optional
Specifies what figure is saved to file. If not specified, the active figure is saved. If a Figure instance is provided, this figure is
saved. If an int is specified, the figure instance to save is looked up by number.

class matplotlib.backends.backend_pdf.Reference(id)
    Bases: object

    PDF reference object. Use PdfFile.reserveObject() to create References.

class matplotlib.backends.backend_pdf.Stream(id, len, file, extra=None)
    Bases: object

    PDF stream object.

    This has no pdfRepr method. Instead, call begin(), then output the contents of the stream by calling write(), and finally call end().

    id: object id of stream; len: an unused Reference object for the length of the stream, or None (to use a memory buffer); file: a PdfFile; extra: a dictionary of extra key-value pairs to include in the stream header

    end()
        Finalize stream.

    write(data)
        Write some data on the stream.

matplotlib.backends.backend_pdf.fill(strings, linelen=75)
    Make one string from sequence of strings, with whitespace in between. The whitespace is chosen to form lines of at most linelen characters, if possible.

matplotlib.backends.backend_pdf.new_figure_manager(num, *args, **kwargs)
    Create a new figure manager instance

matplotlib.backends.backend_pdf.new_figure_manager_given_figure(num, figure)
    Create a new figure manager instance for the given figure.

matplotlib.backends.backend_pdf.pdfRepr(obj)
    Map Python objects to PDF syntax.
42.1 matplotlib.cbook

A collection of utility functions and classes. Originally, many (but not all) were from the Python Cookbook – hence the name cbook.

This module is safe to import from anywhere within matplotlib; it imports matplotlib only at runtime.

class matplotlib.cbook.Bunch(**kwds)

Often we want to just collect a bunch of stuff together, naming each item of the bunch; a dictionary’s OK for that, but a small do-nothing class is even handier, and prettier to use. Whenever you want to group a few variables:

>>> point = Bunch(datum=2, squared=4, coord=12)
>>> point.datum

By: Alex Martelli
From: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52308

class matplotlib.cbook.CallbackRegistry(*args)

Handle registering and disconnecting for a set of signals and callbacks:

>>> def oneat(x):
...    print('eat', x)
>>> def ondrink(x):
...    print('drink', x)

>>> from matplotlib.cbook import CallbackRegistry
>>> callbacks = CallbackRegistry()

>>> id_eat = callbacks.connect('eat', oneat)
>>> id_drink = callbacks.connect('drink', ondrink)

>>> callbacks.process('drink', 123)
drink 123
>>> callbacks.process('eat', 456)
eat 456
In practice, one should always disconnect all callbacks when they are no longer needed to avoid dangling references (and thus memory leaks). However, real code in matplotlib rarely does so, and due to its design, it is rather difficult to place this kind of code. To get around this, and prevent this class of memory leaks, we instead store weak references to bound methods only, so when the destination object needs to die, the CallbackRegistry won’t keep it alive. The Python stdlib weakref module can not create weak references to bound methods directly, so we need to create a proxy object to handle weak references to bound methods (or regular free functions). This technique was shared by Peter Parente on his “Mindtrove” blog.

```python
callbacks.process('be merry', 456)  # nothing will be called
callbacks.disconnect(id_eat)
callbacks.process('eat', 456)  # nothing will be called
```

`connect(s, func)`

register `func` to be called when a signal `s` is generated `func` will be called

`disconnect(cid)`

disconnect the callback registered with callback id `cid`

`process(s, *args, **kwargs)`

process signal `s`. All of the functions registered to receive callbacks on `s` will be called with `*args` and `**kwargs`

```python
class matplotlib.cbook.GetRealpathAndStat

class matplotlib.cbook.Grouper(init=[])  
Bases: object

This class provides a lightweight way to group arbitrary objects together into disjoint sets when a full-blown graph data structure would be overkill.

Objects can be joined using `join()`, tested for connectedness using `joined()`, and all disjoint sets can be retrieved by using the object as an iterator.

The objects being joined must be hashable and weak-referenceable.

For example:

```python
>>> from matplotlib.cbook import Grouper
>>> class Foo(object):
...     def __init__(self, s):
...         self.s = s
...     def __repr__(self):
...         return self.s
... 
>>> a, b, c, d, e, f = [Foo(x) for x in 'abcdef']
>>> grp = Grouper()
>>> grp.join(a, b)
>>> grp.join(b, c)
>>> grp.join(d, e)
>>> sorted(map(tuple, grp))
[(a, b, c), (d, e)]
```
>>> grp.joined(a, b)
True
>>> grp.joined(a, c)
True
>>> grp.joined(a, d)
False

clean()

Clean dead weak references from the dictionary
get_siblings(a)

Returns all of the items joined with a, including itself.
join(a, *args)

Join given arguments into the same set. Accepts one or more arguments.
joined(a, b)

Returns True if a and b are members of the same set.

class matplotlib.cbook.Idle(func)

Bases: matplotlib.cbook.Scheduler

Schedule callbacks when scheduler is idle
run()

waittime = 0.05

eXception matplotlib.cbook.MatplotlibDeprecationWarning

Bases: exceptions.UserWarning

A class for issuing deprecation warnings for Matplotlib users.

In light of the fact that Python builtin DeprecationWarnings are ignored by default as of Python 2.7 (see link below), this class was put in to allow for the signaling of deprecation, but via UserWarnings which are not ignored by default.

http://docs.python.org/dev/whatsnew/2.7.html#the-future-for-python-2-x

class matplotlib.cbook.MemoryMonitor(nmax=20000)

clear()

plot(i0=0, isub=1, fig=None)

report(segments=4)

xy(i0=0, isub=1)
class matplotlib.cbook.Null(*args, **kwargs)
    Null objects always and reliably “do nothing.”

class matplotlib.cbook.RingBuffer(size_max)
    class that implements a not-yet-full buffer

    append(x)
        append an element at the end of the buffer

    get()
        Return a list of elements from the oldest to the newest.

class matplotlib.cbook.Scheduler
    Bases: threading.Thread

    Base class for timeout and idle scheduling

    id = 0

    idlelock = <thread.lock object at 0x7f8bfd6b0ed0>

    stop()

class matplotlib.cbook.Sorter
    Sort by attribute or item

    Example usage:

    sort = Sorter()

    list = [(1, 2), (4, 8), (0, 3)]
    dict = [{'a': 3, 'b': 4}, {'a': 5, 'b': 2}, {'a': 0, 'b': 0},
            {'a': 9, 'b': 9}]

    sort(list)  # default sort
    sort(list, 1)  # sort by index 1
    sort(dict, 'a')  # sort a list of dicts by key 'a'

    byAttribute(data, attributename, inplace=1)

    byItem(data, itemindex=None, inplace=1)

    sort(data, itemindex=None, inplace=1)

class matplotlib.cbook.Stack(default=None)
    Bases: object

    Implement a stack where elements can be pushed on and you can move back and forth. But no pop.
    Should mimic home / back / forward in a browser
back()
    move the position back and return the current element

bubble(o)
    raise o to the top of the stack and return o. o must be in the stack

clear()
    empty the stack

empty()

forward()
    move the position forward and return the current element

home()
    push the first element onto the top of the stack

push(o)
    push object onto stack at current position - all elements occurring later than the current position are discarded

remove(o)
    remove element o from the stack

class matplotlib.cbook.Timeout(wait, func)
    Bases: matplotlib.cbook.Scheduler
    Schedule recurring events with a wait time in seconds

    run()

class matplotlib.cbook.Xlator
    Bases: dict
    All-in-one multiple-string-substitution class

    Example usage:

    text = "Larry Wall is the creator of Perl"
    adict = {
        "Larry Wall" : "Guido van Rossum",
        "creator" : "Benevolent Dictator for Life",
        "Perl" : "Python",
    }

    print multiple_replace(adict, text)

    xlat = Xlator(adict)
    print xlat.xlat(text)

    xlat(text)
    Translate text, returns the modified text.
matplotlib.cbook.align_iterators(func, *iterables)

This generator takes a bunch of iterables that are ordered by func It sends out ordered tuples:

(func(row), [rows from all iterators matching func(row)])

It is used by matplotlib.mlab.recs_join() to join record arrays

matplotlib.cbook.allequal(seq)

Return True if all elements of seq compare equal. If seq is 0 or 1 length, return True

matplotlib.cbook.allpairs(x)

return all possible pairs in sequence x

Condensed by Alex Martelli from this thread on c.l.python

matplotlib.cbook.alltrue(seq)

Return True if all elements of seq evaluate to True. If seq is empty, return False.

matplotlib.cbook.boxplot_stats(X, whis=1.5, bootstrap=None, labels=None)

Returns list of dictionaries of statistics to be use to draw a series of box and whisker plots. See the Returns section below to the required keys of the dictionary. Users can skip this function and pass a user- defined set of dictionaries to the new axes.bxp method instead of relying on MPL to do the calcs.

Parameters X : array-like
    Data that will be represented in the boxplots. Should have 2 or fewer dimensions.

whis : float, string, or sequence (default = 1.5)  
    As a float, determines the reach of the whiskers past the first and third quartiles (e.g., Q3 + whis*IQR, QR = interquartile range, Q3-Q1). Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max data. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, can whis be the string ‘range’ to force the whiskers to the min and max of the data. In the edge case that the 25th and 75th percentiles are equivalent, whis will be automatically set to ‘range’

bootstrap : int or None (default)
    Number of times the confidence intervals around the median should be bootstrapped (percentile method).

labels : sequence
    Labels for each dataset. Length must be compatible with dimensions of X

Returns bxpstats : list of dict
    A list of dictionaries containing the results for each column of data. Keys of each dictionary are the following:
# Boxplot and Notch Parameters

<table>
<thead>
<tr>
<th>Key</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>tick label for the boxplot</td>
</tr>
<tr>
<td>mean</td>
<td>arithmetic mean value</td>
</tr>
<tr>
<td>med</td>
<td>50th percentile</td>
</tr>
<tr>
<td>q1</td>
<td>first quartile (25th percentile)</td>
</tr>
<tr>
<td>q3</td>
<td>third quartile (75th percentile)</td>
</tr>
<tr>
<td>cilo</td>
<td>lower notch around the median</td>
</tr>
<tr>
<td>cihi</td>
<td>upper notch around the median</td>
</tr>
<tr>
<td>whislo</td>
<td>end of the lower whisker</td>
</tr>
<tr>
<td>whishi</td>
<td>end of the upper whisker</td>
</tr>
<tr>
<td>fliers</td>
<td>outliers</td>
</tr>
</tbody>
</table>

## Notes

Non-bootstrapping approach to confidence interval uses Gaussian-based asymptotic approximation:

\[
\text{med} \pm 1.57 \times \frac{\text{iqr}}{\sqrt{N}}
\]


### class matplotlib.cbook.converter

Bases: object

Base class for handling string -> python type with support for missing values

#### is_missing(s)

### matplotlib.cbook.dedent(s)

Remove excess indentation from docstring s.

Discards any leading blank lines, then removes up to n whitespace characters from each line, where n is the number of leading whitespace characters in the first line. It differs from textwrap.dedent in its deletion of leading blank lines and its use of the first non-blank line to determine the indentation.

It is also faster in most cases.

### matplotlib.cbook.delete_masked_points(*args)

Find all masked and/or non-finite points in a set of arguments, and return the arguments with only the unmasked points remaining.

Arguments can be in any of 5 categories:

1. 1-D masked arrays
2. 1-D ndarrays
3. ndarrays with more than one dimension
4. other non-string iterables
5. anything else

The first argument must be in one of the first four categories; any argument with a length differing from that of the first argument (and hence anything in category 5) then will be passed through unchanged.
Masks are obtained from all arguments of the correct length in categories 1, 2, and 4; a point is bad if masked in a masked array or if it is a nan or inf. No attempt is made to extract a mask from categories 2, 3, and 4 if np.isfinite() does not yield a Boolean array.

All input arguments that are not passed unchanged are returned as ndarrays after removing the points or rows corresponding to masks in any of the arguments.

A vastly simpler version of this function was originally written as a helper for Axes.scatter().

```
matplotlib.cbook.deprecated(since, message=u'', name=u'', alternative=u'', pending=False, obj_type=u'function')
```

Decorator to mark a function as deprecated.

**Parameters**

- `since` : str
  The release at which this API became deprecated. This is required.

- `message` : str, optional
  Override the default deprecation message. The format specifier `%func` may be used for the name of the function, and `%alternative` may be used in the deprecation message to insert the name of an alternative to the deprecated function. `%obj_type` may be used to insert a friendly name for the type of object being deprecated.

- `name` : str, optional
  The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

  ```python
def new_function():
  ...
  oldFunction = new_function
```

- `alternative` : str, optional
  An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

- `pending` : bool, optional
  If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

**Examples**

Basic example:

```python
@deprecated('1.4.0')
def the_function_to_deprecate():
  pass
```

```
matplotlib.cbook.dict_delall(d, keys)
delete all of the keys from the dict d
```

994 Chapter 42. cbook
matplotlib.cbook.exception_to_str(s=None)

matplotlib.cbook.file_requires_unicode(x)
    Returns True if the given writable file-like object requires Unicode to be written to it.

matplotlib.cbook.finddir(o, match, case=False)
    return all attributes of o which match string in match. if case is True require an exact case match.

matplotlib.cbook.flatten(seq, scalarp=<function is_scalar_or_string at 0x7f8c0238c050>)
    Returns a generator of flattened nested containers

    For example:

>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [['[42, (5, 23)]', ]])
>>> print(list(flatten(l)))
['John', 'Hunter', 1, 23, 42, 5, 23]

By: Composite of Holger Krekel and Luther Blissett From:
http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/121294 and Recipe 1.12 in cookbook

matplotlib.cbook.get_recursive_filelist(args)
    Recurse all the files and dirs in args ignoring symbolic links and return the files as a list of strings

matplotlib.cbook.get_sample_data(fname, asfileobj=True)
    Return a sample data file. fname is a path relative to the mpl-data/sample_data directory. If
asfileobj is True return a file object, otherwise just a file path.

    Set the rc parameter examples.directory to the directory where we should look, if sample_data files
are stored in a location different than default (which is 'mpl-data/sample_data' at the same level of
'matplotlib' Python module files).

    If the filename ends in .gz, the file is implicitly ungzipped.

matplotlib.cbook.get_split_ind(seq, N)
    seq is a list of words. Return the index into seq such that:

    len(' '.join(seq[:ind]))<=N

matplotlib.cbook.is_math_text(s)

matplotlib.cbook.is_numlike(obj)
    return true if obj looks like a number

matplotlib.cbook.is_scalar(obj)
    return true if obj is not string like and is not iterable

matplotlib.cbook.is_scalar_or_string(val)
    Return whether the given object is a scalar or string like.
matplotlib.cbook.is_sequence_of_strings(obj)
Returns true if obj is iterable and contains strings

matplotlib.cbook.is_string_like(obj)
Return True if obj looks like a string

matplotlib.cbook.is_writable_file_like(obj)
return true if obj looks like a file object with a write method

matplotlib.cbook.issubclass_safe(x, klass)
return issubclass(x, klass) and return False on a TypeError

matplotlib.cbook.iterable(obj)
return true if obj is iterable

matplotlib.cbook.listFiles(root, patterns=’*’, recurse=1, return_folders=0)
Recursively list files
from Parmar and Martelli in the Python Cookbook

class matplotlib.cbook.maxdict(maxsize)
Bases: dict
A dictionary with a maximum size; this doesn’t override all the relevant methods to contrain size, just setitem, so use with caution

matplotlib.cbook.mkdirs(newdir, mode=511)
make directory newdir recursively, and set mode. Equivalent to

  > mkdir -p NEWDIR
  > chmod MODE NEWDIR

matplotlib.cbook.mplDeprecation
alias of MatplotlibDeprecationWarning

matplotlib.cbook.onetru(seq)
Return True if one element of seq is True. It seq is empty, return False.

matplotlib.cbook.pieces(seq, num=2)
Break up the seq into num tuples

matplotlib.cbook.popall(seq)
empty a list

matplotlib.cbook.print_cycles(objects, outstream=<open file ‘<stdout>’, mode ‘w’ at 0x7f8c03e32150>, show_progress=False)

  objects A list of objects to find cycles in. It is often useful to pass in gc.garbage to find the cycles that
  are preventing some objects from being garbage collected.
  outstream The stream for output.
  show_progress If True, print the number of objects reached as they are found.

matplotlib.cbook.recursive_remove(path)
matplotlib.cbook.report_memory(i=0)
    return the memory consumed by process

matplotlib.cbook.restrict_dict(d, keys)
    Return a dictionary that contains those keys that appear in both d and keys, with values from d.

matplotlib.cbook.reverse_dict(d)
    reverse the dictionary – may lose data if values are not unique!

matplotlib.cbook.safe_masked_invalid(x)

matplotlib.cbook.safezip(*args)
    make sure args are equal len before zipping

class matplotlib.cbook.silent_list(type, seq=None)
    Bases: list
    override repr when returning a list of matplotlib artists to prevent long, meaningless output. This is
meant to be used for a homogeneous list of a given type

matplotlib.cbook.simple_linear_interpolation(a, steps)

matplotlib.cbook.soundex(name, len=4)
    soundex module conforming to Odell-Russell algorithm

matplotlib.cbook.strip_math(s)
    remove latex formatting from mathtext

matplotlib.cbook.to_filehandle(fname, flag=u'rU', returnOpened=False)
    fname can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends
in .gz. flag is a read/write flag for file()

class matplotlib.cbook.todate(fmt=u'%Y-%m-%d', missing=u'Null', missingval=None)
    Bases: matplotlib.cbook.converter
    convert to a date or None
    use a time.strptime() format string for conversion

class matplotlib.cbook.todatetime(fmt=u'%Y-%m-%d', missing=u'Null', missingval=None)
    Bases: matplotlib.cbook.converter
    convert to a datetime or None
    use a time.strptime() format string for conversion

class matplotlib.cbook.tofloat(missing=u'Null', missingval=None)
    Bases: matplotlib.cbook.converter
    convert to a float or None

class matplotlib.cbook.toint(missing=u'Null', missingval=None)
    Bases: matplotlib.cbook.converter
    convert to an int or None
class matplotlib.cbook.tostr(missing='Null', missingval='')
    Bases: matplotlib.cbook.converter
    convert to string or None

matplotlib.cbook.unicode_safe(s)

matplotlib.cbook.unique(x)
    Return a list of unique elements of x

matplotlib.cbook.unmasked_index_ranges(mask, compressed=True)
    Find index ranges where mask is False.
    mask will be flattened if it is not already 1-D.

    Returns Nx2 numpy.ndarray with each row the start and stop indices for slices of the compressed numpy.ndarray corresponding to each of N uninterrupted runs of unmasked values. If optional argument compressed is False, it returns the start and stop indices into the original numpy.ndarray, not the compressed numpy.ndarray. Returns None if there are no unmasked values.

    Example:

    y = ma.array(np.arange(5), mask = [0,0,1,0,0])
    ii = unmasked_index_ranges(ma.getmaskarray(y))
    # returns array [[0,2], [2,4]]
    y.compressed()[ii[1,0]:ii[1,1]]
    # returns array [3,4]
    ii = unmasked_index_ranges(ma.getmaskarray(y), compressed=False)
    # returns array [[0, 2], [3, 5]]
    y.filled()[ii[1,0]:ii[1,1]]
    # returns array [3,4]

    Prior to the transforms refactoring, this was used to support masked arrays in Line2D.

matplotlib.cbook.violin_stats(X, method, points=100)
    Returns a list of dictionaries of data which can be used to draw a series of violin plots. See the Returns section below to view the required keys of the dictionary. Users can skip this function and pass a user-defined set of dictionaries to the axes.vplot method instead of using MPL to do the calculations.

    Parameters X: array-like
        Sample data that will be used to produce the gaussian kernel density estimates. Must have 2 or fewer dimensions.

    method: callable
        The method used to calculate the kernel density estimate for each column of data. When called via method(v, coords), it should return a vector of the values of the KDE evaluated at the values specified in coords.

    points: scalar, default = 100
Defines the number of points to evaluate each of the gaussian kernel density estimates at.

**Returns** A list of dictionaries containing the results for each column of data.

The dictionaries contain at least the following:
- `coords`: A list of scalars containing the coordinates this particular kernel density estimate was evaluated at.
- `vals`: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in `coords`.
- `mean`: The mean value for this column of data.
- `median`: The median value for this column of data.
- `min`: The minimum value for this column of data.
- `max`: The maximum value for this column of data.

`matplotlib.cbook.warn_deprecated(since, message=u'', name=u'', alternative=u'', pending=False, obj_type=u'attribute')`

Used to display deprecation warning in a standard way.

**Parameters**

- `since` : str
  The release at which this API became deprecated.

- `message` : str, optional
  Override the default deprecation message. The format specifier `%func` may be used for the name of the function, and `%alternative`s may be used in the deprecation message to insert the name of an alternative to the deprecated function. `%obj_type` may be used to insert a friendly name for the type of object being deprecated.

- `name` : str, optional
  The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

  ```python
def new_function():
    ...
oldFunction = new_function
```

- `alternative` : str, optional
  An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

- `pending` : bool, optional
  If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

- `obj_type` : str, optional
  The object type being deprecated.

**Examples**

Basic example:
# To warn of the deprecation of "matplotlib.name_of_module"
warn_deprecated('1.4.0', name='matplotlib.name_of_module',
                 obj_type='module')

matplotlib.cbook.wrap(prefix, text, cols)
wrap text with prefix at length cols
43.1 matplotlib.cm

This module provides a large set of colormaps, functions for registering new colormaps and for getting a colormap by name, and a mixin class for adding color mapping functionality.

class matplotlib.cm.ScalarMappable(norm=None, cmap=None)
   This is a mixin class to support scalar data to RGBA mapping. The ScalarMappable makes use of data normalization before returning RGBA colors from the given colormap.

   Parameters
   norm : matplotlib.colors.Normalize instance
      The normalizing object which scales data, typically into the interval [0, 1].
   cmap : str or Colormap instance
      The colormap used to map normalized data values to RGBA colors.

   add_checker(checker)
      Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

   autoscale()
      Autoscale the scalar limits on the norm instance using the current array

   autoscale_None()
      Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

   changed()
      Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

   check_update(checker)
      If mappable has changed since the last check, return True; else return False

   cmap = None
      The Colormap instance of this ScalarMappable.

   colorbar = None
      The last colorbar associated with this ScalarMappable. May be None.

   get_array()
      Return the array
get_clim()
return the min, max of the color limits for image scaling

get_cmap()
return the colormap
	norm = None
The Normalization instance of this ScalarMappable.

set_array(A)
Set the image array from numpy array A

set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_cmap(cmap)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_colorbar(*args, **kwargs)
Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set_norm(norm)
set the normalization instance

to_rgba(x, alpha=None, bytes=False)
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

matplotlib.cm.get_cmap(name=None, lut=None)
Get a colormap instance, defaulting to rc values if name is None.

Colormaps added with register_cmap() take precedence over built-in colormaps.
If `name` is a `matplotlib.colors.Colormap` instance, it will be returned.

If `lut` is not None it must be an integer giving the number of entries desired in the lookup table, and `name` must be a standard mpl colormap name with a corresponding data dictionary in `datad`.

`matplotlib.cm.register_cmap(name=None, cmap=None, data=None, lut=None)`

Add a colormap to the set recognized by `get_cmap()`.

It can be used in two ways:

```python
register_cmap(name='swirly', cmap=swirly_cmap)
register_cmap(name='choppy', data=choppydata, lut=128)
```

In the first case, `cmap` must be a `matplotlib.colors.Colormap` instance. The `name` is optional; if absent, the name will be the `name` attribute of the `cmap`.

In the second case, the three arguments are passed to the `LinearSegmentedColormap` initializer, and the resulting colormap is registered.

`matplotlib.cm.revcmap(data)`

Can only handle specification `data` in dictionary format.
44.1 `matplotlib.collections`

Classes for the efficient drawing of large collections of objects that share most properties, e.g., a large number of line segments or polygons.

The classes are not meant to be as flexible as their single element counterparts (e.g., you may not be able to select all line styles) but they are meant to be fast for common use cases (e.g., a large set of solid line segments)

```python
class matplotlib.collections.AsteriskPolygonCollection(numsides, rotation=0, sizes=(1,), **kwargs)
```

Bases: `matplotlib.collections.RegularPolyCollection`

Draw a collection of regular asterisks with `numsides` points.

- **numsides** the number of sides of the polygon
- **rotation** the rotation of the polygon in radians
- **sizes** gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:
- **edgecolors**: None
- **facecolors**: None
- `linewdiths`: None
- `antialiaseds`: None
- `offsets`: None
- `transOffset`: `transforms.IdentityTransform()`
- `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
- `cmap`: None (optional for `matplotlib.cm.ScalarMappable`)

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets)

If any of `edgecolors`, `facecolors`, `linewdiths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

Example: see `examples/dynamic_collection.py` for complete example:

```python
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0, 0, 0, 1)

collection = RegularPolyCollection(
    numsides=5,  # a pentagon
    rotation=0, sizes=(50, ),
    facecolors=facecolors,
    edgecolors=(black, ),
    linewidths=(1, ),
    offsets=offsets,
    transOffset=ax.transData,
)

add_callback(func)

Add a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

`add_checker(checker)`

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

`aname = u'Artist'`

`autoscale()`

Autoscale the scalar limits on the norm instance using the current array

`autoscale_None()`

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`changed()`

Call this whenever the mappable is changed to notify all the callback listeners to the 'changed' signal

`check_update(checker)`

If mappable has changed since the last check, return True; else return False

`contains(mouseevent)`

Test whether the mouse event occurred in the collection.
Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

**convert_xunits**(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

**convert_yunits**(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

**draw**(artist, renderer, *args, **kwargs)

**findobj**(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.

**get_agg_filter**()
return filter function to be used for agg filter

**get_alpha**()
Return the alpha value used for blending - not supported on all backends

**get_animated**()
Return the artist’s animated state

**get_array**()
Return the array

**get_axes**()
Return the Axes instance the artist resides in, or None

**get_children**()
Return a list of the child Artist's this : class:`Artist ` contains.

**get_cmap**()
return the colormap

**get_clim**()
return the min, max of the color limits for image scaling

**get_clip_box**()
Return artist clipbox

**get_clip_on**()
Return whether artist uses clipping

**get_clip_path**()
Return artist clip path

**get_cmap**()
return the colormap

**get_contains**()
Return the _contains test used by the artist, or None for default.
get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
  Return the Figure instance the artist belongs to.

get_gid()
  Returns the group id

get_hatch()
  Return the current hatching pattern

get_label()
  Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_numsides()

get_offset_position()
  Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
  Return the offsets for the collection.
get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()

get_rasterized()
    return True if the artist is to be rasterized

get_rotation()

get_sizes()
    Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.
    Returns sizes : array
    The ‘area’ of each element.

get_sketch_params()
    Returns the sketch parameters for the artist.
    Returns sketch_params : tuple or None
    A 3-tuple with the following elements:
    • scale: The amplitude of the wiggle perpendicular to the source line.
    • length: The length of the wiggle along the line.
    • randomness: The scale factor by which the length is shrunken or expanded.
    May return None if no sketch parameters were set.

get_snap()
    Returns the snap setting which may be:
    • True: snap vertices to the nearest pixel center
    • False: leave vertices as-is
    • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
    Only supported by the Agg and MacOSX backends.

get_transform()
    Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()}
```
get_url()

    Returns the url

get_urls()  

get_visible()

    Return the artist’s visibility

get_window_extent(renderer)

get_zorder()

    Return the Artist’s zorder.

have_units()

    Return True if units are set on the x or y axes

hitlist(event)

    List the children of the artist which contain the mouse event event.

is_figure_set()

    Returns True if the artist is assigned to a Figure.

is_transform_set()

    Returns True if Artist has a transform explicitly set.

pchanged()

    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)

    call signature:

        pick(mouseevent)

        each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()

    Return True if Artist is pickable.

properties()

    return a dictionary mapping property name -> value for all Artist props

remove()

    Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

    Note: relim() will not see collections even if the collection was added to axes with autolim = True.

    Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)

    Remove a callback based on its id.
```
See also:

add_callback() For adding callbacks

**set(**kwargs**)

A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)

set agg_filter function.

set_alpha(alpha)

Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)

Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)

Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)

alias for set_antialiased

set_array(A)

Set the image array from numpy array A

set_axes(axes)

Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)

set the norm limits for image scaling; if vmin is a length 2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(clipbox)

Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(path, transform=None)

Set the artist’s clip path, which may be:

• a Patch (or subclass) instance
• a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.

• None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPPTS: [(Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data
ACCEPPTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.
ACCEPPTS: matplotlib color arg or sequence of rgba tuples

See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_colorbar(*args, **kwargs)
Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPPTS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPPTS: matplotlib color arg or sequence of rgba tuples

set_edgecolors(c)
alias for set_edgecolor
**set_facecolor**

Set the facecolor(s) of the collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_facecolors**

alias for set_facecolor

**set_figure**

Set the *Figure* instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

**set_gid**

Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch**

Set the hatching pattern

*hatch* can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
. - dots
* - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: [ '/', '\', '|', '-', '+', 'x', 'o', 'O', '.' | '*' ]

**set_label**

Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_linestyle**

Set the linestyle(s) for the collection.

ACCEPTS: ['solid' | 'dashed', ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) ]
set_linestyles($ls$)
alias for set linestyle

set_linewidth($lw$)
Set the linewidth(s) for the collection. $lw$ can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths($lw$)
alias for set_linewidth

set_lod($on$)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

set_lw($lw$)
alias for set_linewidth

set_norm($norm$)
set the normalization instance

set_offset_position($offset_position$)
Set how offsets are applied. If $offset_position$ is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If $offset_position$ is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets($offs$)
Set the offsets for the collection. $offs$ can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects($path$)
set path effects, which should be a list of instances of matplotlib.path.PathEffects._Base class or its derivatives.

set_paths()

set_picker($picker$)
Set the epsilon for picking used by this artist

$picker$ can be one of the following:

• None: picking is disabled for this artist (default)
• A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
• A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
• A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:
hit, props = picker(artist, mouseevent)

to determine the hit test. if the mouse event is over the artist, return $hit=True$ and props
is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sizes**(sizes, dpi=72.0)
Set the sizes of each member of the collection.

**Parameters**

- **sizes**: ndarray or None
  The size to set for each element of the collection. The value is
  the ‘area’ of the element.
- **dpi**: float
  The dpi of the canvas. Defaults to 72.0.

**set_sketch_params**(scale=None, length=None, randomness=None)
Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in
  pixels. If scale is None, or not provided, no sketch filter will be
  provided.
- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)
- **randomness**: float, optional
  The scale factor by which the length is shrunken or expanded
  (default 16.0)

**set_snap**(snap)

Sets the snap setting which may be:

- **True**: snap vertices to the nearest pixel center
- **False**: leave vertices as-is
- **None**: (auto) If the path contains only rectilinear line segments, round to the nearest pixel
  center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

**set_url**(url)
Sets the url for the artist
ACCEPTS: a url string

```python
set_urls(urls)
```

**set_visible(**)`b`**
Set the artist’s visibility.

ACCEPTS: [True | False]

```python
set_zorder(**level**)  
```
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

```python
to_rgba(**x**, **alpha=None**, **bytes=False**)  
```
Return a normalized rgba array corresponding to \(x\).

In the normal case, \(x\) is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If \(x\) is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the `alpha` kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the `alpha` kwarg is ignored; it does not replace the pre-existing alpha. A `ValueError` will be raised if the third dimension is other than 3 or 4.

In either case, if `bytes` is `False` (default), the rgba array will be floats in the 0-1 range; if it is `True`, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as \(x\) being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

```python
update(**props**)  
```
Update the properties of this Artist from the dictionary `prop`.

```python
update_from(**other**)  
```
Copy properties from `other` to self

```python
update_scalarmappable()  
```
If the scalar mappable array is not none, update colors from scalar data

```python
zorder = 0
```

**class** `matplotlib.collections.BrokenBarHCollection(**xranges**, **yrange**, **kwargs**)`

Bases: `matplotlib.collections.PolyCollection`

A collection of horizontal bars spanning \(yrange\) with a sequence of \(xranges\).

- `xranges` sequence of \((xmin, xwidth)\)
- `yrange` \((ymin, ywidth)\)

Valid Collection keyword arguments:
- `edgecolors`: None

Chapter 44. collections
• `facecolors`: None
• `linewidths`: None
• `antialiaseds`: None
• `offsets`: None
• `transOffset`: transforms.IdentityTransform()
• `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
• `cmap`: None (optional for `matplotlib.cm.ScalarMappable`)

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets)

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

`add_callback(func)`
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an `id` that is useful for removing the callback with `remove_callback()` later.

`add_checker(checker)`
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

`aname = u'Artist'`

`autoscale()`
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None()`
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`changed()`
Call this whenever the mappable is changed to notify all the callback listeners to the `changed` signal

`check_update(checker)`
If mappable has changed since the last check, return True; else return False

`contains(mouseevent)`
Test whether the mouse event occurred in the collection.

Returns `True` | `False`, `dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits(x)`
For artists in an axes, if the xaxis has units support, convert `x` using xaxis unit type

`convert_yunits(y)`
For artists in an axes, if the yaxis has units support, convert `y` using yaxis unit type

`draw(artist, renderer, *args, **kwargs)`

`findobj(match=None, include_self=True)`
Find artist objects.

Recursively find all `Artist` instances contained in self.

`match` can be
None: return all objects contained in artist.
function with signature boolean = match(artist) used to filter matches
class instance: e.g., Line2D. Only return artists of class type.
If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None

get_children()
Return a list of the child Artist's this :class:'Artist contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or None for default.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()
get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is
    applied after the master transform has been applied, that is, the offsets are in screen coordinates.
    If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are
    in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()

get_rasterized()
    return True if the artist is to be rasterized

get_sizes()
    Returns the sizes of the elements in the collection. The value represents the ‘area’ of the
element.

Returns sizes : array
The ‘area’ of each element.

get_sketch_params()
Returns the sketch parameters for the artist.
Returns sketch_params : tuple or None
A 3-tuple with the following elements:
• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.
May return None if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
Only supported by the Agg and MacOSX backends.

get_transform()
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

g_get_visible()
Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
Return the Artist’s zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.
is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    call signature:

    pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

    Note: relim() will not see collections even if the collection was added to axes with autolim = True.

    Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
    Remove a callback based on its id.

    See also:

    add_callback() For adding callbacks

set(**kwargs)
    A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
    set agg_filter fuction.

set_alpha(alpha)
    Set the alpha transparencies of the collection. alpha must be a float or None.

    ACCEPTS: float or None

set_animated(b)
    Set the artist’s animation state.

    ACCEPTS: [True | False]
**set_antialiased** *(aa)*
Set the antialiasing state for rendering.

*ACCEPTS: Boolean or sequence of booleans*

**set_antialiaseds** *(aa)*
alias for **set_antialiased**

**set_array** *(A)*
Set the image array from numpy array A

**set_axes** *(axes)*
Set the **Axes** instance in which the artist resides, if any.

*ACCEPTS: an **Axes** instance*

**set_clim** *(vmin=None, vmax=None)*
set the norm limits for image scaling; if **vmin** is a length 2 sequence, interpret it as (**vmin**, **vmax**) which is used to support setp

*ACCEPTS: a length 2 sequence of floats*

**set_clip_box** *(clipbox)*
Set the artist’s clip **Bbox**.

*ACCEPTS: a **matplotlib.transforms.Bbox** instance*

**set_clip_on** *(b)*
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

*ACCEPTS: [True | False]*

**set_clip_path** *(path, transform=None)*
Set the artist’s clip path, which may be:

- a **Path** (or subclass) instance
- a **Path** instance, in which case an optional **Transform** instance may be provided, which will be applied to the path before using it for clipping.
- **None**, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to **None**.

*ACCEPTS: [ (Path, Transform) | Patch | None ]*

**set_cmap** *(cmap)*
set the colormap for luminance data

*ACCEPTS: a colormap or registered colormap name*

**set_color** *(c)*
Set both the edgecolor and the facecolor.

*ACCEPTS: matplotlib color arg or sequence of rgba tuples*

See also:

**set_facecolor()**, **set_edgecolor()** For setting the edge or face color individually.
**set_colorbar(*args, **kwargs)**

Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

**set_contains(picker)**

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return $hit = True$ and $props$ is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**set_dashes(ls)**

alias for set_linestyle

**set_edgecolor(c)**

Set the edgecolor(s) of the collection. $c$ can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_edgecolors(c)**

alias for set_edgecolor

**set_facecolor(c)**

Set the facecolor(s) of the collection. $c$ can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_facecolors(c)**

alias for set_facecolor

**set_figure(fig)**

Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid(gid)**

Sets the (group) id for the artist

ACCEPTS: an id string
**set_hatch(hatch)**
Set the hatching pattern

*hatch* can be one of:

/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\'] | '|' | '+-' | '+' | 'x' | 'o' | 'O' | '.' | '*' ]

**set_label(s)**
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_linestyle(ls)**
Set the linestyle(s) for the collection.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted'] | (offset, on-off-dash-seq) ]

**set_linestyles(ls)**
alias for set_linestyle

**set_linewidth(lw)**
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

**set_linewidths(lw)**
alias for set_linewidth

**set_lod(on)**
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

**set_lw(lw)**
alias for set_linewidth
**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(path_effects)
set path_effects, which should be a list of instances of matplotlib.path.Affine2D or its derivatives.

**set_paths**(verts, closed=True)
This allows one to delay initialization of the vertices.

**set_picker**(picker)
Set the epsilon for picking used by this artist

picker can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  
  hit, props = picker(artist, mouseevent)

  to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sizes**(sizes, dpi=72.0)
Set the sizes of each member of the collection.

Parameters sizes: ndarray or None
The size to set for each element of the collection. The value is the ‘area’ of the element.

**dpi** : float
The dpi of the canvas. Defaults to 72.0.

**set_sketch_params** *(scale=None, length=None, randomness=None)*
Sets the sketch parameters.

**Parameters**

- **scale** : float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

- **length** : float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness** : float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)
Set the Transform instance used by this artist.

**ACCEPTS:** Transform instance

**set_url**(url)
Sets the url for the artist

**ACCEPTS:** a url string

**set_urls**(urls)

**set_verts**(verts, closed=True)
This allows one to delay initialization of the vertices.

**set_visible**(b)
Set the artist’s visibility.

**ACCEPTS:** [True | False]

**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

**ACCEPTS:** any number

**static span_where**(x, ymin, ymax, where, **kwargs)
Create a BrokenBarHCollection to plot horizontal bars from over the regions in x where where is True. The bars range on the y-axis from ymin to ymax
A **BrokenBarHCollection** is returned. **kwargs** are passed on to the collection.

**to_rgba**(*x*, **alpha=None, bytes=False*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the **alpha** kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the **alpha** kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if **bytes** is **False** (default), the rgba array will be floats in the 0-1 range; if it is **True**, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update**(**props**)  
Update the properties of this **Artist** from the dictionary **prop**.

**update_from**(**other**)  
copy properties from other to self

**update_scalarmappable()**  
If the scalar mappable array is not none, update colors from scalar data

**zorder = 0**

**class** **matplotlib.collections.CircleCollection**(*sizes, **kwargs*)

Bases: **matplotlib.collections._CollectionWithSizes**

A collection of circles, drawn using splines.

**sizes** Gives the area of the circle in points^2

Valid Collection keyword arguments:

- **edgecolors**: None  
- **facecolors**: None  
- **linewidths**: None  
- **antialiaseds**: None  
- **offsets**: None  
- **transOffset**: transforms.IdentityTransform()  
- **norm**: None (optional for **matplotlib.cm.ScalarMappable**)  
- **cmap**: None (optional for **matplotlib.cm.ScalarMappable**)  

**offsets** and **transOffset** are used to translate the patch after rendering (default no offsets)

If any of **edgecolors**, **facecolors**, **linewidths**, **antialiaseds** are None, they default to their **matplotlib.rcParams** patch setting, in sequence form.
add_callback(func)
    Adds a callback function that will be called whenever one of the Artist's properties changes.
    Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
    Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = u'Artist'

autoscale()
    Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
    Autoscale the scalar limits on the norm instance using the current array, changing only limits
    that are None

changed()
    Call this whenever the mappable is changed to notify all the callback listeners to the
    'changed' signal

check_update(checker)
    If mappable has changed since the last check, return True; else return False

contains(mouseevent)
    Test whether the mouse event occurred in the collection.
    Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.
    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.
    If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends
get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None

get_children()
    Return a list of the child Artist's this :class:`Artist contains.

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_dashes()

getdatalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern
get_label()
  Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
  Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
  Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
  Return the picker object used by this artist

get_pickradius()

get_rasterized()
  return True if the artist is to be rasterized

get_sizes()
  Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.
    Returns sizes : array
      The ‘area’ of each element.

get_sketch_params()
  Returns the sketch parameters for the artist.
    Returns sketch_params : tuple or None
      A 3-tuple with the following elements:
        • scale: The amplitude of the wiggle perpendicular to the source line.
        • length: The length of the wiggle along the line.
- **randomness**: The scale factor by which the length is shrunken or expanded.

  May return `None` if no sketch parameters were set.

`get_snap()`

Returns the snap setting which may be:

- `True`: snap vertices to the nearest pixel center
- `False`: leave vertices as-is
- `None`: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

`get_transform()`

Return the `Transform` instance used by this artist.

`get_transformed_clip_path_and_affine()`

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms()`

`get_url()`

Returns the url

`get_urls()`

`get_visible()`

Return the artist’s visibility

`get_window_extent(renderer)`

`get_zorder()`

Return the Artist’s zorder.

`have_units()`

Return `True` if units are set on the `x` or `y` axes

`hitlist(event)`

List the children of the artist which contain the mouse event `event`.

`is_figure_set()`

Returns `True` if the artist is assigned to a `Figure`.

`is_transform_set()`

Returns `True` if `Artist` has a transform explicitly set.

`pchanged()`

Fire an event when property changed, calling all of the registered callbacks.

`pick(mouseevent)`

call signature:
pick(mouseevent)

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
Return True if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:
add_callback() For adding callbacks

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha tranparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased

set_array(A)
Set the image array from numpy array A

set_axes(axe)
Set the Axes instance in which the artist resides, if any.
**set_clim** *(vmin=None, vmax=None)*

Set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as \((vmin, vmax)\) which is used to support *setp*

ACCEPTS: a length 2 sequence of floats

**set_clip_box** *(clipbox)*

Set the artist’s clip Bbox.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

**set_clip_on** *(b)*

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**set_clip_path** *(path, transform=None)*

Set the artist’s clip path, which may be:

- a `Patch` (or subclass) instance
- a `Path` instance, in which case an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- `None`, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

ACCEPTS: [(Path, Transform) | Patch | None ]

**set_cmap** *(cmap)*

Set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

**set_color** *(c)*

Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

**set_facecolor()**, **set_edgecolor()** For setting the edge or face color individually.

**set_colorbar** *(*args, **kwargs)*

Deprecated since version 1.3: The `set_colorbar` function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

**set_contains** *(picker)*

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```
If the mouse event is over the artist, return \( \text{hit} = \text{True} \) and \( \text{props} \) is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**set_dashes(**\( ls \)**)

alias for set_linestyle

**set_edgecolor(**\( c \)**)

Set the edgecolor(s) of the collection. \( c \) can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If \( c \) is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_edgecolors(**\( c \)**)

alias for set_edgecolor

**set_facecolor(**\( c \)**)

Set the facecolor(s) of the collection. \( c \) can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If \( c \) is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_facecolors(**\( c \)**)

alias for set_facecolor

**set_figure(**\( fig \)**)

Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid(**\( gid \)**)

Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch(**\( hatch \)**)

Set the hatching pattern

\( hatch \) can be one of:

\[
/ \quad \text{- diagonal hatching} \\
\backslash \quad \text{- back diagonal} \\
| \quad \text{- vertical} \\
- \quad \text{- horizontal} \\
+ \quad \text{- crossed} \\
x \quad \text{- crossed diagonal} \\
o \quad \text{- small circle} \\
o \quad \text{- large circle}
\]
Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: [‘/’ | ‘\’ | ‘|’ | ‘.’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

**set_label**(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_linestyle**(ls)
Set the linestyle(s) for the collection.

ACCEPTS: [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) ]

**set_linestyles**(ls)
alias for set_linestyle

**set_linewidth**(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: float or sequence of floats

**set_linewidths**(lw)
alias for set_linewidth

**set_lod**(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly.

ACCEPTS: [True | False]

**set_lw**(lw)
alias for set_linewidth

**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats
**set_path_effects**(*path_effects*)

set_path_effects, which should be a list of instances of matplotlib.pathdeffect._Base class or its derivatives.

**set_paths()**

**set_picker**(*picker*)

Set the epsilon for picking used by this artist

*picker* can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if *picker* is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and *props* is a dictionary of properties you want added to the PickEvent attributes.

  ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(*pr*)

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to *None*, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sizes**(*sizes, dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

- **sizes**: ndarray or None
  - The size to set for each element of the collection. The value is the ‘area’ of the element.
- **dpi**: float
  - The dpi of the canvas. Defaults to 72.0.

**set_sketch_params**(*scale=None, length=None, randomness=None*)

Sets the the sketch parameters.

**Parameters**

- **scale**: float, optional
  - The amplitude of the wiggle perpendicular to the source line, in pixels. If *scale* is *None*, or not provided, no sketch filter will be provided.
- **length**: float, optional
The length of the wiggle along the line, in pixels (default 128.0)

**randomness** : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

**set_url**(url)
Sets the url for the artist.

ACCEPTS: a url string

**set_urls**(urls)

**set_visible**(b)
Set the artist’s visibility.

ACCEPTS: [True | False]

**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**to_rgba**(x, alpha=None, bytes=False)
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.
update(props)
    Update the properties of this Artist from the dictionary prop.

update_from(other)
    copy properties from other to self

update_scalarmappable()
    If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.Collection(edgecolors=None, facecolors=None, linewidths=None, linestyles='solid', antialiaseds=None, offsets=None, transOffset=None, norm=None, cmap=None, pick_radius=5.0, hatch=None, urls=None, offset_position='screen', zorder=1, **kwargs)

Bases: matplotlib.artist.Artist, matplotlib.cm.ScalarMappable

Base class for Collections. Must be subclassed to be usable.

All properties in a collection must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith element of the collection is:

prop[i % len(props)]

Keyword arguments and default values:
  *edgecolors*: None
  *facecolors*: None
  *linewidths*: None
  *linestyles*: 'solid'
  *antialiaseds*: None
  *offsets*: None
  *transOffset*: transforms.IdentityTransform()
  *offset_position*: 'screen' (default) or 'data'
  *norm*: None (optional for matplotlib.cm.ScalarMappable)
  *cmap*: None (optional for matplotlib.cm.ScalarMappable)
  *hatch*: None
  *zorder*: 1

offsets and transOffset are used to translate the patch after rendering (default no offsets). If offset_position is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.

The use of ScalarMappable is optional. If the ScalarMappable matrix _A is not None (i.e., a call to set_array has been made), at draw time a call to scalar mappable will be made to set the face colors.

Create a Collection

%(Collection)s
add_callback(func)
Add a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = u'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

changed()
Call this whenever the mappable is changed to notify all the callback listeners to the 'changed' signal

check_update(checker)
If mappable has changed since the last check, return True; else return False

contains(mouseevent)
Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
Find artist objects.

Recursively find all Artist instances contained in self.

match can be

• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends
get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None

get_children()
    Return a list of the child Artist’s this :class:`Artist <Artist>` contains.

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern
get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()

get_rasterized()
    return True if the artist is to be rasterized

get_sketch_params()
    Returns the sketch parameters for the artist.

    Returns sketch_params : tuple or None

    A 3-tuple with the following elements:
    • scale: The amplitude of the wiggle perpendicular to the source line.
    • length: The length of the wiggle along the line.
    • randomness: The scale factor by which the length is shrunken or expanded.

    May return None if no sketch parameters were set.

get_snap()
    Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
    Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
    Returns the url

get_urls()

generate visible()
    Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
    Return the Artist’s zorder.

have_units()
    Return True if units are set on the x or y axes

hitlist(event)
    List the children of the artist which contain the mouse event event.

is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    call signature:

    pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.
properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible un-
til the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call
matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim =
True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:
add_callback() For adding callbacks

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased

set_array(A)
Set the image array from numpy array A

set_axes(axes)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, 
vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats
set_clip_box(clipbox)
Set the artist’s clip Bbox.
ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
ACCEPTS: [(Path, Transform) | Patch | None]

set_cmap(cmap)
set the colormap for luminance data
ACCEPTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.
ACCEPTS: matplotlib color arg or sequence of rgba tuples
See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_colorbar(*args, **kwargs)
Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.
set the colorbar and axes instances associated with mappable

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)
If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.
ACCEPTS: a callable function

set_dashes(ls)
alias for set_linestyle
**set_edgecolor**(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_edgecolors**(*c*)
alias for set_edgecolor

**set_facecolor**(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_facecolors**(*c*)
alias for set_facecolor

**set_figure**(*fig*)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid**(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch**(*hatch*)
Set the hatching pattern

*hatch* can be one of:

/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
O  - large circle
.  - dots
*  - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.
Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: [‘/’ | ‘\’ | ‘|’ | ‘-’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

set_linestyle(ls)
Set the linestyle(s) for the collection.

ACCEPTS: [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) ]

set_linestyles(ls)
alias for set_linestyle

set_linewidth(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(lw)
alias for set_linewidth

set_lod(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

set_lw(lw)
alias for set_linewidth

set_norm(norm)
set the normalization instance

set_offset_position(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(path_effects)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths()
**set_picker**(*picker*)
Set the epsilon for picking used by this artist

*picker* can be one of the following:
- **None**: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
def picker(artist, mouseevent):
    hit, props = picker(artist, mouseevent)
```
to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(*pr*)

**set_rasterized**(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sketch_params**(*scale=None, length=128.0, randomness=16.0*)
Sets the sketch parameters.

Parameters

- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is *None*, or not provided, no sketch filter will be provided.

- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(*snap*)
Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(*t*)
Set the *Transform* instance used by this artist.
ACCEPTS: \texttt{Transform} instance

\texttt{set_url}(url)

Sets the url for the artist

ACCEPTS: a url string

\texttt{set_urls}(urls)

\texttt{set_visible}(b)

Set the artist's visibility.

ACCEPTS: [True \mid False]

\texttt{set_zorder}(level)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

\texttt{to_rgba}(x, alpha=None, bytes=False)

Return a normalized rgba array corresponding to \(x\).

In the normal case, \(x\) is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If \(x\) is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the \texttt{alpha} kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the \texttt{alpha} kwarg is ignored; it does not replace the pre-existing alpha. A \texttt{ValueError} will be raised if the third dimension is other than 3 or 4.

In either case, if \texttt{bytes} is \texttt{False} (default), the rgba array will be floats in the 0-1 range; if it is \texttt{True}, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as \(x\) being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

\texttt{update}(props)

Update the properties of this \texttt{Artist} from the dictionary \texttt{prop}.

\texttt{update_from}(other)

copy properties from other to self

\texttt{update_scalarmappable}()

If the scalar mappable array is not none, update colors from scalar data

\texttt{zorder = 0}

class \texttt{matplotlib.collections.EllipseCollection}(widths, heights, angles, units='points', **kwargs)

Bases: \texttt{matplotlib.collections.Collection}

A collection of ellipses, drawn using splines.
**widths**: sequence lengths of first axes (e.g., major axis lengths)

**heights**: sequence lengths of second axes

**angles**: sequence angles of first axes, degrees CCW from the X-axis

**units**: ['points', 'inches', 'dots', 'width', 'height', 'x', 'y', 'xy']

Units in which majors and minors are given; ‘width’ and ‘height’ refer to the dimensions of the axes, while ‘x’ and ‘y’ refer to the offsets data units. ‘xy’ differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the `Ellipse` with axes.transData as its transform.

Additional kwargs inherited from the base **Collection**:

Valid Collection keyword arguments:

- **edgecolors**: None
- **facecolors**: None
- **linewdths**: None
- **antialiaseds**: None
- **offsets**: None
- **transOffset**: transforms.IdentityTransform()
- **norm**: None (optional for matplotlib.cm.ScalarMappable)
- **cmap**: None (optional for matplotlib.cm.ScalarMappable)

Offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.

**add_callback(func)**

Adds a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with `remove_callback()` later.

**add_checker(checker)**

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

**aname = u’Artist’**

**autoscale()**

 Autoscale the scalar limits on the norm instance using the current array

**autoscale_None()**

 Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**changed()**

 Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

**check_update(checker)**

 If mappable has changed since the last check, return True; else return False

**contains(mouseevent)**

 Test whether the mouse event occurred in the collection.

 Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.
convert_xunits($x$)
For artists in an axes, if the xaxis has units support, convert $x$ using xaxis unit type

convert_yunits($y$)
For artists in an axes, if the yaxis has units support, convert $y$ using yaxis unit type

draw($artist$, renderer, *args, **kwargs)

findobj($match=None$, $include_self=True$)
Find artist objects.
Recursively find all Artist instances contained in self.

match can be
• None: return all objects contained in artist.
• function with signature boolean = match($artist$) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.
If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None

get_children()
Return a list of the child Artist's this :class:`Artist contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or None for default.
get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()
get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()

get_rasterized()
    return True if the artist is to be rasterized

get_sketch_params()
    Returns the sketch parameters for the artist.
    Returns sketch_params : tuple or None
    A 3-tuple with the following elements:
    • scale: The amplitude of the wiggle perpendicular to the source line.
    • length: The length of the wiggle along the line.
    • randomness: The scale factor by which the length is shrunken or expanded.
    May return None if no sketch parameters were set.

get_snap()
    Returns the snap setting which may be:
    • True: snap vertices to the nearest pixel center
    • False: leave vertices as-is
    • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
    Only supported by the Agg and MacOSX backends.

get_transform()
    Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
    Returns the url

get_urls()

get_visible()
    Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
    Return the Artist’s zorder.
have_units()
    Return True if units are set on the x or y axes

hitlist(event)
    List the children of the artist which contain the mouse event event.

is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    call signature:

        pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible un-
    til the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call
    matplotlib.axes.Axes.relim() to update the axes limits if desired.

    Note: relim() will not see collections even if the collection was added to axes with autolim =
    True.

    Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
    Remove a callback based on its id.

    See also:

    add_callback() For adding callbacks

set(**kwargs)
    A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
    set agg_filter function.

set_alpha(alpha)
    Set the alpha tranparencies of the collection. alpha must be a float or None.

    ACCEPTS: float or None
**set_animated**(*b*)  
Set the artist’s animation state.  

ACCEPTS: [True | False]

**set_antialiased**(*aa*)  
Set the antialiasing state for rendering.  

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds**(*aa*)  
alias for set_antialiased

**set_array**(*A*)  
Set the image array from numpy array *A*

**set_axes**(*axes*)  
Set the Axes instance in which the artist resides, if any.  

ACCEPTS: an Axes instance

**set_clim**(*vmin=None, vmax=None*)  
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as (*vmin*, *vmax*) which is used to support setp  

ACCEPTS: a length 2 sequence of floats

**set_clip_box**(*clipbox*)  
Set the artist’s clip Bbox.  

ACCEPTS: a matplotlib.transforms.Bbox instance

**set_clip_on**(*b*)  
Set whether artist uses clipping.  

When False artists will be visible out side of the axes which can lead to unexpected results.  

ACCEPTS: [True | False]

**set_clip_path**(*path, transform=None*)  
Set the artist’s clip path, which may be:  
- a Patch (or subclass) instance  
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.  
- None, to remove the clipping path  

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.  

ACCEPTS: [(Path, Transform) | Patch | None ]

**set_cmap**(*cmap*)  
set the colormap for luminance data  

ACCEPTS: a colormap or registered colormap name

**set_color**(*c*)  
Set both the edgecolor and the facecolor.
ACCEPETS: matplotlib color arg or sequence of rgba tuples

See also:

set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_colorbar(*args, **kwargs)
Depreciated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use
the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function
which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you
want returned with the contains test.

ACCEPETS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color arg (all patches have same
color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the
sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch
boundary will not be drawn.

ACCEPETS: matplotlib color arg or sequence of rgba tuples

set_edgecolors(c)
alias for set_edgecolor

set_facecolor(c)
Set the facecolor(s) of the collection. c can be a matplotlib color arg (all patches have same
color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the
sequence.

If c is ‘none’, the patch will not be filled.

ACCEPETS: matplotlib color arg or sequence of rgba tuples

set_facecolors(c)
alias for set_facecolor

set_figure(fig)
Set the Figure instance the artist belongs to.

ACCEPETS: a matplotlib.figure.Figure instance
**set_gid(gid)**

Sets the (group) id for the artist

**ACCEPTS:** an id string

**set_hatch(hatch)**

Set the hatching pattern

*hatch* can be one of:

* / - diagonal hatching
* \ - back diagonal
* | - vertical
* - - horizontal
* + - crossed
* x - crossed diagonal
* o - small circle
* O - large circle
* . - dots
* * - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**ACCEPTS:** ['/' | '\' | '|' | '+-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

**set_label(s)**

Set the label to *s* for auto legend.

**ACCEPTS:** string or anything printable with ‘%s’ conversion.

**set_linestyle(ls)**

Set the linestyle(s) for the collection.

**ACCEPTS:** ['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq)]

**set_linestyles(ls)**

alias for set_linestyle

**set_linewidth(lw)**

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**ACCEPTS:** float or sequence of floats

**set_linewidths(lw)**

alias for set_linewidth

**set_lod(on)**

Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly
ACCEP'TS: [True | False]

**set_lw(lw)**
alias for **set_linewidth**

**set_norm(norm)**
set the normalization instance

**set_offset_position(offset_position)**
Set how offsets are applied. If offset_position is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets(offsets)**
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEP'TS: float or sequence of floats

**set_path_effects(path_effects)**
set path_effects, which should be a list of instances of matplotlib.path.effect._Base class or its derivatives.

**set_paths()**

**set_picker(picker)**
Set the epsilon for picking used by this artist

picker can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  hit, props = picker(artist, mouseevent)

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEP'TS: [None|float|boolean|callable]

**set_pickradius(pr)**

**set_rasterized(rasterized)**
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior
set_sketch_params(scale=None, length=None, randomness=None)

Sets the sketch parameters.

**Parameters**

- `scale`: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If `scale` is `None`, or not provided, no sketch filter will be provided.

- `length`: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- `randomness`: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(t)

Set the Transform instance used by this artist.

ACCEPTS: Transform instance

set_url(url)

Sets the url for the artist

ACCEPTS: a url string

set_urls(urls)

set_visible(b)

Set the artist’s visibility.

ACCEPTS: [True | False]

set_zorder(level)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

to_rgba(x, alpha=None, bytes=False)

Return a normalized rgba array corresponding to `x`.

In the normal case, `x` is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If `x` is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the `alpha` kwarg (defaulting to 1) will be used to fill in the transparency. If
the last dimension is 4, the \texttt{alpha} kwarg is ignored; it does not replace the pre-existing alpha. A
ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if \texttt{bytes} is \texttt{False} (default), the rgba array will be floats in the 0-1 range; if it is
\texttt{True}, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as \texttt{x}
being a masked rgba array, or being an integer type other than uint8, or being a floating point
rgba array with values outside the 0-1 range.

\texttt{update(props)}
Update the properties of this \texttt{Artist} from the dictionary \texttt{prop}.

\texttt{update_from(other)}
copy properties from other to self

\texttt{update_scalarmappable()}
If the scalar mappable array is not none, update colors from scalar data

\texttt{zorder = 0}

\texttt{class matplotlib.collections.EventCollection(positions, orientation=None, lineoffset=0,
linelength=1, linewidth=None, color=None, linestyle='solid', antialiased=None,
**kwargs)}
Bases: \texttt{matplotlib.collections.LineCollection}

A collection of discrete events.

An event is a 1-dimensional value, usually the position of something along an axis, such as time or
length. Events do not have an amplitude. They are displayed as v
\texttt{positions} a sequence of numerical values or a 1D numpy array. Can be None
\texttt{orientation} \texttt{[‘horizontal’ | ‘vertical’ | None]} defaults to ‘horizontal’ if not specified or None
\texttt{lineoffset} a single numerical value, corresponding to the offset of the center of the markers from the
origin
\texttt{linelength} a single numerical value, corresponding to the total height of the marker (i.e. the marker
stretches from lineoffset+linelength/2 to lineoffset-linelength/2). Defaults to 1
\texttt{linewidth} a single numerical value
\texttt{color} must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).
\texttt{linestyle} \texttt{[‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ]}
\texttt{antialiased} 1 or 2

If \texttt{linewidth}, \texttt{color}, or \texttt{antialiased} is None, they default to their rcParams setting, in sequence form.
\texttt{norm} None (optional for \texttt{matplotlib.cm.ScalarMappable})
\texttt{cmap} None (optional for \texttt{matplotlib.cm.ScalarMappable})
\texttt{pickradius} is the tolerance for mouse clicks picking a line. The default is 5 pt.

The use of \texttt{ScalarMappable} is optional. If the \texttt{ScalarMappable} array _A is not None (i.e., a call to
\texttt{set_array()} has been made), at draw time a call to scalar mappable will be made to set the colors.

\textbf{Example:}
add_callback(func)
   Adds a callback function that will be called whenever one of the Artist’s properties changes.
   Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
   Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

add_positions(position)
   add one or more events at the specified positions

aname = u’Artist’

append_positions(position)
   add one or more events at the specified positions

autoscale()
   Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
   Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

changed()
   Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal
check_update(checker)
    If mappable has changed since the last check, return True; else return False

color(c)
    Set the color(s) of the line collection. c can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence

    ACCEPTS: matplotlib color arg or sequence of rgba tuples

contains(mouseevent)
    Test whether the mouse event occurred in the collection.

    Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

extend_positions(position)
    add one or more events at the specified positions

findobj(match=None, include_self=True)
    Find artist objects.

    Recursively find all Artist instances contained in self.

    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.

    If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None

get_children()
    Return a list of the child Artist's this :class:'Artist contains.
get_clim()  
return the min, max of the color limits for image scaling

get_clip_box()  
Return artist clipbox

get_clip_on()  
Return whether artist uses clipping

get_clip_path()  
Return artist clip path

get_cmap()  
return the colormap

get_color()  
get the color of the lines used to mark each event

get_colors()  

get_contains()  
Return the _contains test used by the artist, or None for default.

get_dashes()  

get_datalim(transData)

get_edgecolor()  

get_edgecolors()  

get_facecolor()  

get_facecolors()  

get_figure()  
Return the Figure instance the artist belongs to.

get_gid()  
Returns the group id

get_hatch()  
Return the current hatching pattern

get_label()  
Get the label used for this artist in the legend.

get_linelength()  
get the length of the lines used to mark each event
get_lineoffset()  
get the offset of the lines used to mark each event

get_linestyle()  
get the style of the lines used to mark each event ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_linestyles()  

get_linewidth()  
get the width of the lines used to mark each event

get_linewidths()  

get_offset_position()  
Returns how offsets are applied for the collection. If offset_position is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()  

get_offsets()  
Return the offsets for the collection.

get_orientation()  
get the orientation of the event line, may be: [ 'horizontal' | 'vertical' ]

get_path_effects()  

get_paths()  

get_picker()  
Return the picker object used by this artist

get_pickradius()  

get_positions()  
return an array containing the floating-point values of the positions

get_rasterized()  
return True if the artist is to be rasterized

get_segments()  

get_sketch_params()  
Returns the sketch parameters for the artist.
Returns sketch_params : tuple or None  
A 3-tuple with the following elements:
• **scale**: The amplitude of the wiggle perpendicular to the source line.
• **length**: The length of the wiggle along the line.
• **randomness**: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

**get_snap()**

Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**get_transform()**

Return the `Transform` instance used by this artist.

**get_transformed_clip_path_and_affine()**

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms()**

**get_url()**

Returns the url

**get_urls()**

**get_visible()**

Return the artist’s visibility

**get_window_extent(renderer)**

**get_zorder()**

Return the `Artist`'s zorder.

**have_units()**

Return `True` if units are set on the x or y axes

**hitlist(event)**

List the children of the artist which contain the mouse event `event`

**is_figure_set()**

Returns `True` if the artist is assigned to a `Figure`.

**is_horizontal()**

True if the eventcollection is horizontal, False if vertical

**is_transform_set()**

Returns `True` if `Artist` has a transform explicitly set.
pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
call signature:

pick(mouseevent)

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
Return True if Artist is pickable.

properties()
return a dictionary mapping property name - value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:
add_callback() For adding callbacks

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha tranparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased
set_array(A)
Set the image array from numpy array A

set_axes(axes)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(clipbox)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:

• a Patch (or subclass) instance
• a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
• None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [(Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(c)
Set the color(s) of the line collection. c can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_colorbar(**kwargs)
Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable
set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_edgecolors(c)
alias for set_edgecolor

set_facecolor(c)
Set the facecolor(s) of the collection. c can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_facecolors(c)
alias for set_facecolor

set_figure(fig)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

set_gid(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(hatch)
Set the hatching pattern

hatch can be one of:
Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*']

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

set_linelength(linelength)
set the length of the lines used to mark each event

set_lineoffset(lineoffset)
set the offset of the lines used to mark each event

set_linestyle(ls)
Set the linestyle(s) for the collection.

ACCEPTS: ['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq)]

set_linestyles(ls)
alias for set_linestyle

set_linewidth(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(lw)
alias for set_linewidth

set_lod(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

set_lw(lw)
alias for set_linewidth
**set_norm**(*norm*)
set the normalization instance

**set_offset_position**(*offset_position*)
Set how offsets are applied. If *offset_position* is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If *offset_position* is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_orientation**(*orientation=None*)
set the orientation of the event line [‘horizontal’ | ‘vertical’ | None] defaults to ‘horizontal’ if not specified or None

**set_path_effects**(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.path_effects._Base class or its derivatives.

**set_paths**(*segments*)

**set_picker**(*picker*)
Set the epsilon for picking used by this artist

*picker* can be one of the following:

- None: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(*pr*)

**set_positions**(*positions*)
set the positions of the events to the specified value

**set_rasterized**(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.
Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_segments**(segments)

**set_sketch_params**(scale=None, length=None, randomness=None)

Sets the sketch parameters.

**Parameters**

- **scale** : float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length** : float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness** : float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)

Set the **Transform** instance used by this artist.

ACCEPTS: **Transform** instance

**set_url**(url)

Sets the url for the artist

ACCEPTS: a url string

**set_urls**(urls)

**set_verts**(segments)

**set_visible**(b)

Set the artist’s visibility.

ACCEPTS: [True | False]

**set_zorder**(level)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**switch_orientation**()

switch the orientation of the event line, either from vertical to horizontal or vice versa
to_rgba(x, alpha=None, bytes=False)

Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(props)

Update the properties of this Artist from the dictionary prop.

update_from(other)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.LineCollection(segments, linewidths=None, colors=None, antialiaseds=None, linestyles='solid', offsets=None, transOffset=None, norm=None, cmap=None, pickradius=5, zorder=2, **kwargs)

Bases: matplotlib.collections.Collection

All parameters must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith line segment is:

prop[i % len(props)]

i.e., the properties cycle if the len of props is less than the number of segments.

segments a sequence of (line0, line1, line2), where:

\[ \text{linen} = (x0, y0), (x1, y1), \ldots (xm, ym) \]

or the equivalent numpy array with two columns. Each line can be a different length.

colors must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

antialiaseds must be a sequence of ones or zeros

linestyles ['solid' | ‘dashed’ | ‘dashdot’ | ‘dotted’] a string or dash tuple. The dash tuple is:
(offset, onoffseq),

where onoffseq is an even length tuple of on and off ink in points. If lineoffsets, linecolors, or antialiaseds is None, they default to their rcParams setting, in sequence form.

If offsets and transOffset are not None, then offsets are transformed by transOffset and applied after the segments have been transformed to display coordinates.

If offsets is not None but transOffset is None, then the offsets are added to the segments before any transformation. In this case, a single offset can be specified as:

offsets=(xo,yo)

and this value will be added cumulatively to each successive segment, so as to produce a set of successively offset curves.

norm None (optional for matplotlib.cm.ScalarMappable)
cmap None (optional for matplotlib.cm.ScalarMappable)
pickradius is the tolerance for mouse clicks picking a line. The default is 5 pt.
zorder The zorder of the LineCollection. Default is 2
The use of ScalarMappable is optional. If the ScalarMappable array _A is not None (i.e., a call to set_array() has been made), at draw time a call to scalar mappable will be made to set the colors.

add_callback(func)
   Adds a callback function that will be called whenever one of the Artist's properties changes.
   Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
   Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = u'Artist'

autoscale()
   Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
   Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

changed()
   Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(checker)
   If mappable has changed since the last check, return True; else return False

color(c)
   Set the color(s) of the line collection. c can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence
   ACCEPTS: matplotlib color arg or sequence of rgba tuples
contains(mouseevent)
Test whether the mouse event occurred in the collection.
Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.
match can be
• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.
If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None

get_children()
return a list of the child Artist's this :class:`Artist contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

44.1. matplotlib.collections 1073
get_color()

get_colors()

get_contains()

    Return the _contains test used by the artist, or None for default.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()

    Return the Figure instance the artist belongs to.

get_gid()

    Returns the group id

get_hatch()

    Return the current hatching pattern

get_label()

    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()

    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.
get_offset_transform()

def get_offset_transform()

get_offsetsets()

Return the offsets for the collection.

def get_offsetsets()

get_path_effects()

get_paths()

get_picker()

Return the picker object used by this artist

def get_picker()

get_pickradius()

get_rasterized()

return True if the artist is to be rasterized

def get_rasterized()

get_segments()

get_sketch_params()

Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None

A 3-tuple with the following elements:

• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

get_snap()

Returns the snap setting which may be:

• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

def get_snap()

get_transform()

Return the Transform instance used by this artist.

def get_transform()

get_transformed_clip_path_and_affine()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

def get_transformed_clip_path_and_affine()

get_transforms()

get_url()

Returns the url

def get_url()
get_urls()

get_visible()
    Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
    Return the Artist’s zorder.

have_units()
    Return True if units are set on the x or y axes

hitlist(event)
    List the children of the artist which contain the mouse event event.

is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    call signature:

        pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
    Remove a callback based on its id.

See also:

add_callback() For adding callbacks
set(**kwargs)
    A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
    set agg_filter fuction.

set_alpha(alpha)
    Set the alpha tranparencies of the collection. alpha must be a float or None.
    ACCEPTS: float or None

set_animated(b)
    Set the artist’s animation state.
    ACCEPTS: [True | False]

set_antialiased(aa)
    Set the antialiasing state for rendering.
    ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
    alias for set_antialiased

set_array(A)
    Set the image array from numpy array A

set_axes(axes)
    Set the Axes instance in which the artist resides, if any.
    ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
    set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp
    ACCEPTS: a length 2 sequence of floats

set_clip_box(clipbox)
    Set the artist’s clip Bbox.
    ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
    Set whether artist uses clipping.
    When False artists will be visible out side of the axes which can lead to unexpected results.
    ACCEPTS: [True | False]

set_clip_path(path, transform=None)
    Set the artist’s clip path, which may be:
    • a Patch (or subclass) instance
    • a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
    • None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [(Path, Transform) | Patch | None]

`set_cmap(cmap)`
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

`set_color(c)`
Set the color(s) of the line collection. `c` can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

`set_colorbar(*args, **kwargs)`
Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

`set_contains(picker)`
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return `hit = True` and `props` is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

`set_dashes(ls)`
alias for set_linestyle

`set_edgecolor(c)`
Set the edgecolor(s) of the collection. `c` can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

`set_edgecolors(c)`
alias for set_edgecolor

`set_facecolor(c)`
Set the facecolor(s) of the collection. `c` can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘none’, the patch will not be filled.
ACCEP|TS: matplotlib color arg or sequence of rgba tuples

**set_facecolors**(*c*)
alias for set_facecolor

**set_figure**(*fig*)
Set the Figure instance the artist belongs to.

ACCEP|TS: a matplotlib.figure.Figure instance

**set_gid**(*gid*)
Sets the (group) id for the artist

ACCEP|TS: an id string

**set_hatch**(*hatch*)
Set the hatching patte|n

*hatch* can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
. - dots
* - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEP|TS: [ ' '/'\'|'|'|'|'|'|'|'|'|'|'|'|'.|'*' ]

**set_label**(*s*)
Set the label to s for auto legend.

ACCEP|TS: string or anything printable with ‘%s’ conversion.

**set_linestyle**(*ls*)
Set the linestyle(s) for the collection.

ACCEP|TS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) ]

**set_linestyles**(*ls*)
alias for set_linestyle

**set_linewidth**(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence
ACCEP|S: float or sequence of floats

**set_linewidths**(lw)
alias for set_linewidth

**set_lod**(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEP|S: [True | False]

**set_lw**(lw)
alias for set_linewidth

**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEP|S: float or sequence of floats

**set_path_effects**(path_effects)
set path_effects, which should be a list of instances of matplotlib.path_effect._Base class or its derivatives.

**set_paths**(segments)

**set_picker**(picker)
Set the epsilon for picking used by this artist

picker can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

\[
\text{hit, props} = \text{picker}(\text{artist, mouseevent})
\]

to determine the hit test. if the mouse event is over the artist, return \text{hit}=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEP|S: [None|float|boolean|callable]
set_pickradius\((pr)\)

set_rasterized\((rasterized)\)
Force rasterized (bitmap) drawing in vector backend output.
Defaults to None, which implies the backend’s default behavior
ACCEPS: [True | False | None]

set_segments\((segments)\)

set_sketch_params\((scale=None, length=None, randomness=None)\)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap\((snap)\)
Sets the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
Only supported by the Agg and MacOSX backends.

set_transform\(t\)
Set the Transform instance used by this artist.
ACCEPS: Transform instance

set_url\(url\)
Sets the url for the artist
ACCEPS: a url string

set_urls\(urls\)

set_verts\(segments\)

set_visible\(b\)
Set the artist’s visibility.
ACCEPS: [True | False]
**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**to_rgba**(x, alpha=None, bytes=False)
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update**(props)
Update the properties of this Artist from the dictionary prop.

**update_from**(other)
copy properties from other to self

**update_scalarmappable**()
If the scalar mappable array is not none, update colors from scalar data

**zorder** = 0

**class** matplotlib.collections.PatchCollection**(patches, match_original=False, **kwargs)**
**Bases:** matplotlib.collections.Collection
A generic collection of patches.
This makes it easier to assign a color map to a heterogeneous collection of patches.
This also may improve plotting speed, since PatchCollection will draw faster than a large number of patches.

patches a sequence of Patch objects. This list may include a heterogeneous assortment of different patch types.
match_original If True, use the colors and linewidths of the original patches. If False, new colors may be assigned by providing the standard collection arguments, facecolor, edgecolor, linewidths, norm or cmap.

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.
The use of ScalarMappable is optional. If the ScalarMappable matrix _A is not None (i.e., a call to set_array has been made), at draw time a call to scalar mappable will be made to set the face colors.

```
add_callback(func)
    Adds a callback function that will be called whenever one of the Artist's properties changes.
    Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
    Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = u'Artist'

autoscale()
    Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
    Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

changed()
    Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(checker)
    If mappable has changed since the last check, return True; else return False

contains(mouseevent)
    Test whether the mouse event occurred in the collection.
    Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.
    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.
    If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
    return filter function to be used for agg filter
get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None

get_children()
    Return a list of the child Artist's this :class:`Artist` contains.

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get.figure()
    Return the Figure instance the artist belongs to.

get_gid()
    Returns the group id
get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.
Returns sketch_params : tuple or None
A 3-tuple with the following elements:
• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.
May return None if no sketch parameters were set.
get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
Return the Artist’s zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
call signature:

pick(mouseevent)

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set
pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible until
    the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call
    matplotlib.axes.Axes.relim() to update the axes limits if desired.

    Note: relim() will not see collections even if the collection was added to axes with autolim =
    True.

    Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
    Remove a callback based on its id.

    See also:
    add_callback() For adding callbacks

set(**kwargs)
    A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
    set agg_filter fuction.

set_alpha(alpha)
    Set the alpha transparencies of the collection. alpha must be a float or None.

    ACCEPTS: float or None

set_animated(b)
    Set the artist’s animation state.

    ACCEPTS: [True | False]

set_antialiased(aa)
    Set the antialiasing state for rendering.

    ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
    alias for set_antialiased

set_array(A)
    Set the image array from numpy array A

set_axes(axes)
    Set the Axes instance in which the artist resides, if any.

    ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
    set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin,
    vmax) which is used to support setp
set_clip_box(clipbox)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
• a Patch (or subclass) instance
• a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
• None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_colorbar(*args, **kwargs)
Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function
**set_dashes**(*ls*)
alias for set linestyle

**set_edgecolor**(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_edgecolors**(*c*)
alias for set_edgecolor

**set_facecolor**(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_facecolors**(*c*)
alias for set_facecolor

**set_figure**(*fig*)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid**(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch**(*hatch*)
Set the hatching pattern

*hatch* can be one of:

/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
O  - large circle
.  - dots
*  - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.
Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: [‘/’| ‘\’| ‘|’| ‘-‘| ‘+’| ‘x’| ‘o’| ‘O’| ‘.’| ‘*’]

**set_label**(s)

Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_linestyle**(ls)

Set the linestyle(s) for the collection.

ACCEPTS: [‘solid’| ‘dashed’, ‘dashdot’, ‘dotted’| (offset, on-off-dash-seq)]

**set_linestyles**(ls)

alias for set_linestyle

**set_linewidth**(lw)

Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

**set_linewidths**(lw)

alias for set_linewidth

**set_lod**(on)

Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

**set_lw**(lw)

alias for set_linewidth

**set_norm**(norm)

set the normalization instance

**set_offset_position**(offset_position)

Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)

Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(path_effects)

set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.
```

**set_paths**(*patches*)

**set_picker**(*picker*)

Set the epsilon for picking used by this artist

picker can be one of the following:
- None: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(*pr*)

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sketch_params**(*scale=None, length=None, randomness=None*)

Sets the the sketch parameters.

Parameters

scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

44.1. matplotlib.collections
**set_transform(t)**
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

**set_url(url)**
Sets the url for the artist

ACCEPTS: a url string

**set_urls(urls)**

**set_visible(b)**
Set the artist’s visibility.

ACCEPTS: [True | False]

**set_zorder(level)**
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**to_rgba(x, alpha=None, bytes=False)**
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update(props)**
Update the properties of this Artist from the dictionary prop.

**update_from(other)**
copy properties from other to self

**update_scalarmappable()**
If the scalar mappable array is not none, update colors from scalar data

**zorder = 0**

**class matplotlib.collections.PathCollection(paths, sizes=None, **kwargs)**
Bases: matplotlib.collections._CollectionWithSizes
This is the most basic `Collection` subclass.

`paths` is a sequence of `matplotlib.path.Path` instances.

Valid Collection keyword arguments:
- `edgecolors`: None
- `facecolors`: None
- `linewdths`: None
- `antialiaseds`: None
- `offsets`: None
- `transOffset`: transforms.IdentityTransform()
- `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
- `cmap`: None (optional for `matplotlib.cm.ScalarMappable`)

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets)

If any of `edgecolors`, `facecolors`, `linewdths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

`add_callback(func)`

Adds a callback function that will be called whenever one of the Artist’s properties changes.

Returns an `id` that is useful for removing the callback with `remove_callback()` later.

`add_checker(checker)`

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

`aname = u’Artist’`

`autoscale()`

Autoscale the scalar limits on the norm instance using the current array

`autoscale_None()`

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`changed()`

Call this whenever the mappable is changed to notify all the callback listeners to the ‘changed’ signal

`check_update(checker)`

If mappable has changed since the last check, return True; else return False

`contains(mouseevent)`

Test whether the mouse event occurred in the collection.

Returns `True | False`, `dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits(x)`

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

`convert_yunits(y)`

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

`draw(artist, renderer, *args, **kwargs)`
**findobj**(match=None, include_self=True)

Find artist objects.

Recursively find all Artist instances contained in self.

*match* can be

- None: return all objects contained in artist.
- function with signature boolean = match(artist) used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

**get_agg_filter**()

return filter function to be used for agg filter

**get_alpha**()

Return the alpha value used for blending - not supported on all backends

**get_animated**()

Return the artist’s animated state

**get_array**()

Return the array

**get_axes**()

Return the Axes instance the artist resides in, or None

**get_children**()

Return a list of the child Artist's this :class:`Artist` contains.

**get_clim**()

return the min, max of the color limits for image scaling

**get_clip_box**()

Return artist clipbox

**get_clip_on**()

Return whether artist uses clipping

**get_clip_path**()

Return artist clip path

**get_cmap**()

return the colormap

**get_contains**()

Return the _contains test used by the artist, or None for default.

**get_dashes**()

**get_datalim**(transData)

**get_edgecolor**()
get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()
get_rasterized()
    return True if the artist is to be rasterized

get_sizes()
    Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.
    Returns sizes : array
    The ‘area’ of each element.

get_sketch_params()
    Returns the sketch parameters for the artist.
    Returns sketch_params : tuple or None
    A 3-tuple with the following elements:
    • scale: The amplitude of the wiggle perpendicular to the source line.
    • length: The length of the wiggle along the line.
    • randomness: The scale factor by which the length is shrunken or expanded.
    May return None if no sketch parameters were set.

get_snap()
    Returns the snap setting which may be:
    • True: snap vertices to the nearest pixel center
    • False: leave vertices as-is
    • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
    Only supported by the Agg and MacOSX backends.

get_transform()
    Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
    Returns the url

get_urls()

get_visible()
    Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
    Return the Artist’s zorder.
have_units()
    Return True if units are set on the x or y axes

hitlist(event)
    List the children of the artist which contain the mouse event event.

is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    call signature:

        pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

    Note: relim() will not see collections even if the collection was added to axes with autolim = True.

    Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
    Remove a callback based on its id.

    See also:

        add_callback() For adding callbacks

set(**kwargs)
    A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
    set agg_filter fuction.

set_alpha(alpha)
    Set the alpha tranparencies of the collection. alpha must be a float or None.

    ACCEPTS: float or None
set_animated\( (b) \)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased\( (aa) \)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds\( (aa) \)
alias for set_antialiased

set_array\( (A) \)
Set the image array from numpy array \( A \)

set_axes\( (axes) \)
Set the \texttt{Axes} instance in which the artist resides, if any.

ACCEPTS: an \texttt{Axes} instance

set_clim\( (vmin=\text{None}, \text{vmax}=\text{None}) \)
set the norm limits for image scaling; if \( \text{vmin} \) is a length2 sequence, interpret it as \( \text{vmin}, \text{vmax} \) which is used to support \texttt{setp}

ACCEPTS: a length 2 sequence of floats

set_clip_box\( (clipbox) \)
Set the artist’s clip \texttt{Bbox}.

ACCEPTS: a \texttt{matplotlib.transforms.Bbox} instance

set_clip_on\( (b) \)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path\( (path, \text{transform}=\text{None}) \)
Set the artist’s clip path, which may be:
- a \texttt{Patch} (or subclass) instance
- a \texttt{Path} instance, \textbf{in which case} an optional \texttt{Transform} instance may be provided, which will be applied to the path before using it for clipping.
- \texttt{None}, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to \texttt{None}.

ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap\( (cmap) \)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color\( (c) \)
Set both the edgecolor and the facecolor.
ACCEP'T$	extbackslash$T$: matplotlib color arg or sequence of rgba tuples

\textbf{See also:}

\texttt{set\_facecolor()}, \texttt{set\_edgecolor()} For setting the edge or face color individually.

\texttt{set\_colorbar}(*\texttt{args}, **\texttt{kwargs})

Deprecated since version 1.3: The \texttt{set\_colorbar} function was deprecated in version 1.3. Use the \texttt{colorbar} attribute instead.

set the colorbar and axes instances associated with mappable

\texttt{set\_contains}(\texttt{picker})

Replace the \texttt{contains} test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

\begin{verbatim}
hit, props = picker(artist, mouseevent)
\end{verbatim}

If the mouse event is over the artist, return $hit = True$ and $props$ is a dictionary of properties you want returned with the \texttt{contains} test.

ACCEP'T$	extbackslash$T$: a callable function

\texttt{set\_dashes}($\texttt{ls}$)

alias for \texttt{set\_linestyle}

\texttt{set\_edgecolor}($c$)

Set the edgecolor(s) of the collection. $c$ can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEP'T$	extbackslash$T$: matplotlib color arg or sequence of rgba tuples

\texttt{set\_edgecolors}($c$)

alias for \texttt{set\_edgecolor}

\texttt{set\_facecolor}($c$)

Set the facecolor(s) of the collection. $c$ can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘none’, the patch will not be filled.

ACCEP'T$	extbackslash$T$: matplotlib color arg or sequence of rgba tuples

\texttt{set\_facecolors}($c$)

alias for \texttt{set\_facecolor}

\texttt{set\_figure}($\texttt{fig}$)

Set the \texttt{Figure} instance the artist belongs to.

ACCEP'T$	extbackslash$T$: a \texttt{matplotlib.figure.Figure} instance
**set_gid**(*gid*)
Sets the (group) id for the artist

**ACCEPTS:** an id string

**set_hatch**(*hatch*)
Set the hatching pattern

*hatch* can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle.
. - dots
* - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**ACCEPTS:** [ '/ ' | '\ ' | '|' | '-.' | '+' | 'x' | 'o' | 'O' | '.' | '*' ]

**set_label**(*s*)
Set the label to *s* for auto legend.

**ACCEPTS:** string or anything printable with ‘%s’ conversion.

**set_linestyle**(*ls*)
Set the linestyle(s) for the collection.

**ACCEPTS:** ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) ]

**set_linestyles**(*ls*)
alias for set_linestyle

**set_linewidth**(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**ACCEPTS:** float or sequence of floats

**set_linewidths**(*lw*)
alias for set_linewidth

**set_lod**(*on*)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly.
**set_lw**(lw)
alias for set_linewidth

**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

**set_path_effects**(path_effects)
set path_effects, which should be a list of instances of matplotlib.pathEffects._Base class or its derivatives.

**set_paths**(paths)

**set_picker**(picker)
Set the epsilon for picking used by this artist

**set_pickradius**(pr)

**set_rasterized**(rasterized)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior
ACCEPES: [True | False | None]

```
set_sizes(sizes, dpi=72.0)
```
Set the sizes of each member of the collection.

**Parameters**

- **sizes**: ndarray or None
  The size to set for each element of the collection. The value is the ‘area’ of the element.
- **dpi**: float
  The dpi of the canvas. Defaults to 72.0.

```
set_sketch_params(scale=None, length=None, randomness=None)
```
Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.
- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)
- **randomness**: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

```
set_snap(snap)
```
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

```
set_transform(t)
```
Set the `Transform` instance used by this artist.

ACCEPES: `Transform` instance

```
set_url(url)
```
Sets the url for the artist

ACCEPES: a url string

```
set_urls(urls)
```

```
set_visible(b)
```
Set the artist’s visiblity.

ACCEPES: [True | False]

```
set_zorder(level)
```
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPES: any number
to_rgba($x$, $alpha$=None, $bytes$=False)

Return a normalized rgba array corresponding to $x$.

In the normal case, $x$ is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If $x$ is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the $alpha$ kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the $alpha$ kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if $bytes$ is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as $x$ being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update($props$)

Update the properties of this Artist from the dictionary $prop$.

update_from($other$)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not none, update colors from scalar data

$zorder = 0$

class matplotlib.collections.PolyCollection($verts$, sizes=None, closed=True, **kwargs)

Bases: matplotlib.collections._CollectionWithSizes

$verts$ is a sequence of ($verts0$, $verts1$, ...) where $verts_i$ is a sequence of $xy$ tuples of vertices, or an equivalent numpy array of shape ($nv$, 2).

$sizes$ is None (default) or a sequence of floats that scale the corresponding $verts_i$. The scaling is applied before the Artist master transform; if the latter is an identity transform, then the overall scaling is such that if $verts_i$ specify a unit square, then $sizes_i$ is the area of that square in points^2. If len($sizes$) < $nv$, the additional values will be taken cyclically from the array.

closed, when True, will explicitly close the polygon.

Valid Collection keyword arguments:

- edgecolors: None
- facecolors: None
- linewidths: None
- antialiaseds: None
- offsets: None
- transOffset: transforms.IdentityTransform()
- norm: None (optional for matplotlib.cm.ScalarMappable)
- cmap: None (optional for matplotlib.cm.ScalarMappable)
offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewights, antialiaseds are None, they default to their
matplotlib.rcParams patch setting, in sequence form.

add_callback(func)
Adds a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = u'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits
that are None

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the
'changed' signal

check_update(checker)
If mappable has changed since the last check, return True; else return False

contains(mouseevent)
Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
Find artist objects.

Recursively find all Artist instances contained in self.

match can be
• None: return all objects contained in artist.
• function with signature boolean = match(artists) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
return filter function to be used for agg filter
get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None

get_children()
Return a list of the child Artist’s this :class:`Artist contains.

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or None for default.

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the Figure instance the artist belongs to.

get_gid()
Returns the group id
get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()

get_rasterized()
    return True if the artist is to be rasterized

get_sizes()
    Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.
    
    Returns sizes : array
        The ‘area’ of each element.

get_sketch_params()
    Returns the sketch parameters for the artist.
    
    Returns sketch_params : tuple or None
        A 3-tuple with the following elements:
• **scale**: The amplitude of the wiggle perpendicular to the source line.
• **length**: The length of the wiggle along the line.
• **randomness**: The scale factor by which the length is shrunk or expanded.

May return `None` if no sketch parameters were set.

**get_snap()**

Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**get_transform()**

Return the `Transform` instance used by this artist.

**get_transformed_clip_path_and_affine()**

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms()**

**get_url()**

Returns the url

**get_urls()**

**get_visible()**

Return the artist’s visibility

**get_window_extent(renderer)**

**get_zorder()**

Return the Artist’s zorder.

**have_units()**

Return `True` if units are set on the `x` or `y` axes

**hitlist(event)**

List the children of the artist which contain the mouse event `event`.

**is_figure_set()**

Returns `True` if the artist is assigned to a `Figure`.

**is_transform_set()**

Returns `True` if `Artist` has a transform explicitly set.

**pchanged()**

Fire an event when property changed, calling all of the registered callbacks.
pick(mouseevent)
call signature:

    pick(mouseevent)

Each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set.

pickable()
Return True if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.Axes.relim() to update the axes limits if desired.

Note: remlim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:
add_callback() For adding callbacks

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased

set_array(A)
Set the image array from numpy array A
set_axes(axes)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(clipbox)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
  • a Patch (or subclass) instance
  • a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
  • None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_colorbar(**args, **kwargs)
Deprecation warning: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:
hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

```
set_dashes(ls)
```
alias for set_linestyle

```
set_edgecolor(c)
```
Set the edgecolor(s) of the collection. c can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

```
set_edgecolors(c)
```
alias for set_edgecolor

```
set_facecolor(c)
```
Set the facecolor(s) of the collection. c can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

```
set_facecolors(c)
```
alias for set_facecolor

```
set_figure(fig)
```
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

```
set_gid(gid)
```
Sets the (group) id for the artist

ACCEPTS: an id string

```
set_hatch(hatch)
```
Set the hatching pattern

hatch can be one of:

/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x - crossed diagonal
o - small circle
O - large circle.
* - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/', '\', '|', '-\', '+', 'x', 'o', 'O', '.', '*']

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(ls)
Set the linestyle(s) for the collection.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (off, on-off-dash-seq) ]

set_linestyles(ls)
alias for set_linestyle

set_linewidth(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: float or sequence of floats

set_linewidths(lw)
alias for set_linewidth

set_lod(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly.

ACCEPTS: [True | False]

set_lw(lw)
alias for set_linewidth

set_norm(norm)
set the normalization instance

set_offset_position(offset_position)
Set how offsets are applied. If offset_position is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.
set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.path_effect._Base class or its derivatives.

set_paths(*verts*, closed=True)
This allows one to delay initialization of the vertices.

set_picker(*picker*)
Set the epsilon for picking used by this artist

*picker* can be one of the following:
- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  hit, props = picker(artist, mouseevent)

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, dpi=72.0)
Set the sizes of each member of the collection.

**Parameters**

- *sizes* : ndarray or None
  The size to set for each element of the collection. The value is the ‘area’ of the element.
- *dpi* : float
  The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None, length=None, randomness=None*)
Sets the the sketch parameters.

**Parameters**

- *scale* : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**: float, optional
   The length of the wiggle along the line, in pixels (default 128.0)

**randomness**: float, optional
   The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
   Sets the snap setting which may be:
   - True: snap vertices to the nearest pixel center
   - False: leave vertices as-is
   - None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
   Only supported by the Agg and MacOSX backends.

**set_transform**(t)
   Set the `Transform` instance used by this artist.

   ACCEPTS: `Transform` instance

**set_url**(url)
   Sets the url for the artist

   ACCEPTS: a url string

**set_urls**(urls)

**set_verts**(verts, closed=True)
   This allows one to delay initialization of the vertices.

**set_visible**(b)
   Set the artist’s visibility.

   ACCEPTS: [True | False]

**set_zorder**(level)
   Set the zorder for the artist. Artists with lower zorder values are drawn first.

   ACCEPTS: any number

**to_rgba**(x, alpha=None, bytes=False)
   Return a normalized rgba array corresponding to x.

   In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

   There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the `alpha` kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the `alpha` kwarg is ignored; it does not replace the pre-existing alpha. A `ValueError` will be raised if the third dimension is other than 3 or 4.
In either case, if `bytes` is `False` (default), the rgba array will be floats in the 0-1 range; if it is `True`, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

```python
update(props)
    Update the properties of this Artist from the dictionary prop.
```

```python
update_from(other)
    copy properties from other to self
```

```python
update_scalarmappable()
    If the scalar mappable array is not none, update colors from scalar data
```

```python
zorder = 0
```

class matplotlib.collections.QuadMesh(meshWidth, meshHeight, coordinates, antialiased=True, shading='flat', **kwargs)

Class for the efficient drawing of a quadrilateral mesh.

A quadrilateral mesh consists of a grid of vertices. The dimensions of this array are \((meshWidth + 1, meshHeight + 1)\). Each vertex in the mesh has a different set of “mesh coordinates” representing its position in the topology of the mesh. For any values \((m, n)\) such that \(0 <= m <= meshWidth\) and \(0 <= n <= meshHeight\), the vertices at mesh coordinates \((m, n)\), \((m, n + 1)\), \((m + 1, n + 1)\), and \((m + 1, n)\) form one of the quadrilaterals in the mesh. There are thus \((meshWidth * meshHeight)\) quadrilaterals in the mesh. The mesh need not be regular and the polygons need not be convex.

A quadrilateral mesh is represented by a \((2 x ((meshWidth + 1) * (meshHeight + 1)))\) numpy array \(coordinates\), where each row is the \(x\) and \(y\) coordinates of one of the vertices. To define the function that maps from a data point to its corresponding color, use the \(set_cmap()\) method. Each of these arrays is indexed in row-major order by the mesh coordinates of the vertex (or the mesh coordinates of the lower left vertex, in the case of the colors).

For example, the first entry in \(coordinates\) is the coordinates of the vertex at mesh coordinates \((0, 0)\), then the one at \((0, 1)\), then at \((0, 2)\) .. \((0, meshWidth)\), \((1, 0)\), \((1, 1)\), and so on.

\(shading\) may be ‘flat’, or ‘gouraud’

```python
add_callback(func)
    Adds a callback function that will be called whenever one of the Artist’s properties changes.
    Returns an id that is useful for removing the callback with remove_callback() later.
```

```python
add_checker(checker)
    Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.
```

```python
aname = u’Artist’
```

```python
autoscale()
    Autoscale the scalar limits on the norm instance using the current array
```
autoscale_None()  
Autoscale the scalar limits on the norm instance using the current array, changing only limits 
that are None

changed()  
Call this whenever the mappable is changed to notify all the callback listeners to the 
‘changed’ signal

check_update(checker)  
If mappable has changed since the last check, return True; else return False

contains(mouseevent)  
Test whether the mouse event occurred in the collection. 
Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

static convert_mesh_to_paths(meshWidth, meshHeight, coordinates)  
Converts a given mesh into a sequence of matplotlib.path.Path objects for easier rendering 
by backends that do not directly support quadmeshes. 
This function is primarily of use to backend implementers.

convert_mesh_to_triangles(meshWidth, meshHeight, coordinates)  
Converts a given mesh into a sequence of triangles, each point with its own color. This is useful 
for experiments using draw_qouraud_triangle.

convert_xunits(x)  
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)  
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)  
Find artist objects. 
Recursively find all Artist instances contained in self. 
match can be
• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches 
• class instance: e.g., Line2D. Only return artists of class type.
If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()  
return filter function to be used for agg filter

get_alpha()  
Return the alpha value used for blending - not supported on all backends

get_animated()  
Return the artist’s animated state
get_array()  
Return the array

get_axes()  
Return the Axes instance the artist resides in, or None

get_children()  
Return a list of the child Artist's this :class:`Artist contains.

get_clim()  
return the min, max of the color limits for image scaling

get_clip_box()  
Return artist clipbox

get_clip_on()  
Return whether artist uses clipping

get_clip_path()  
Return artist clip path

get_cmap()  
return the colormap

get_contains()  
Return the _contains test used by the artist, or None for default.

get_dashes()  

get_datalim(transData)  

get_edgecolor()  

get_edgecolors()  

get_facecolor()  

get_facecolors()  

get_figure()  
Return the Figure instance the artist belongs to.

get_gid()  
Returns the group id

get_hatch()  
Return the current hatching pattern

get_label()  
Get the label used for this artist in the legend.
get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()

Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()

Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()

Return the picker object used by this artist

get_pickradius()

get_rasterized()

return True if the artist is to be rasterized

get_sketch_params()

Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None

A 3-tuple with the following elements:

• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

get_snap()

Returns the snap setting which may be:

• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
Only supported by the Agg and MacOSX backends.

get_transform()
    Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
    Returns the url

get_urls()

get_visible()
    Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
    Return the Artist’s zorder.

have_units()
    Return True if units are set on the x or y axes

hitlist(event)
    List the children of the artist which contain the mouse event event.

is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    call signature:

        pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props
remove()
Remove the artist from the figure if possible. The effect will not be visible until
the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call
matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:
add_callback() For adding callbacks

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha tranparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased

set_array(A)
Set the image array from numpy array A

set_axes(axes)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(clipbox)
Set the artist’s clip Bbox.

44.1. matplotlib.collections
ACCEPTS: a `matplotlib.transforms.Bbox` instance

**set_clip_on(b)**
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**set_clip_path(path, transform=None)**
Set the artist's clip path, which may be:
- a `Patch` (or subclass) instance
- a `Path` instance, in which case an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- `None`, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

ACCEPTS: [(Path, Transform) | Patch | None ]

**set_cmap(cmap)**
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

**set_color(c)**
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:
- `set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

**set_colorbar(*args, **kwargs)**
Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

**set_contains(picker)**
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**set_dashes(ls)**
alias for set_linestyle

**set_edgecolor(c)**
Set the edgecolor(s) of the collection. c can be a matplotlib color arg (all patches have same
color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

```python
set_edgecolors(c)
```

alias for `set_edgecolor`

```python
set_facecolor(c)
```

Set the facecolor(s) of the collection. $c$ can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

```python
set_facecolors(c)
```

alias for `set_facecolor`

```python
set_figure(fig)
```

Set the Figure instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

```python
set_gid(gid)
```

Sets the (group) id for the artist

ACCEPTS: an id string

```python
set_hatch(hatch)
```

Set the hatching pattern

$hatch$ can be one of:

```
/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
O  - large circle
.  - dots
*  - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.
ACCEPTEs: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | ':' | '*' ]

**set_label(s)**
Set the label to s for auto legend.

ACCEPTEs: string or anything printable with '%s' conversion.

**set_linestyle(ls)**
Set the linestyle(s) for the collection.

ACCEPTEs: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) ]

**set_linestyles(ls)**
alias for set_linestyle

**set_linewidth(lw)**
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTEs: float or sequence of floats

**set_linewidths(lw)**
alias for set_linewidth

**set_lod(on)**
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTEs: [True | False]

**set_lw(lw)**
alias for set_linewidth

**set_norm(norm)**
set the normalization instance

**set_offset_position(offset_position)**
Set how offsets are applied. If offset_position is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets(offsets)**
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTEs: float or sequence of floats

**set_path_effects(path_effects)**
set path_effects, which should be a list of instances of matplotlib.path Effect._Base class or its derivatives.

**set_paths()**

**set_picker(picker)**
Set the epsilon for picking used by this artist
picker can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. If the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

  ACCEPTS: [None|float|boolean|callable]

``set_pickradius``(pr)

``set_rasterized``(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

``set_sketch_params``(scale=None, length=None, randomness=None)

Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

``set_snap``(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

``set_transform``(t)

Set the Transform instance used by this artist.

ACCEPTS: Transform instance
Matplotlib, Release 1.4.3

```python
set_url(url)
Sets the url for the artist

ACCEPTS: a url string

set_urls(urls)

set_visible(b)
Set the artist’s visibility.

ACCEPTS: [True | False]

set_zorder(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

to_rgba(x, alpha=None, bytes=False)
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(props)
Update the properties of this Artist from the dictionary prop.

update_from(other)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0
```

class matplotlib.collections.RegularPolyCollection(numsides, rotation=0, sizes=(1,), **kwargs)

Bases: matplotlib.collections._CollectionWithSizes

Draw a collection of regular polygons with numsides.

numsides the number of sides of the polygon
**rotation** the rotation of the polygon in radians

**sizes** gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- **edgecolors**: None
- **facecolors**: None
- **linewidths**: None
- **antialiaseds**: None
- **offsets**: None
- **transOffset**: transforms.IdentityTransform()
- **norm**: None (optional for matplotlib.cm.ScalarMappable)
- **cmap**: None (optional for matplotlib.cm.ScalarMappable)

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets)

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

Example: see examples/dynamic_collection.py for complete example:

```python
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
)
```

**add_callback** *(func)*

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an `id` that is useful for removing the callback with `remove_callback()` later.

**add_checker** *(checker)*

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

**aname** = u'Artist'

**autoscale()**

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None()**

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**changed()**

Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal
check_update(checker)
    If mappable has changed since the last check, return True; else return False

contains(mouseevent)
    Test whether the mouse event occurred in the collection.
    Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.
    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.
    If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None

get_children()
    Return a list of the child Artist's this :class:`Artist contains.

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path
get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_numsides()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.
get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()

get_rasterized()
    return True if the artist is to be rasterized

get_rotation()

get_sizes()
    Returns the sizes of the elements in the collection. The value represents the ‘area’ of the
    element.

    Returns sizes : array
        The ‘area’ of each element.

get_sketch_params()
    Returns the sketch parameters for the artist.

    Returns sketch_params : tuple or None
        A 3-tuple with the following elements:
        • scale: The amplitude of the wiggle perpendicular to the source line.
        • length: The length of the wiggle along the line.
        • randomness: The scale factor by which the length is shrunked or expanded.
        May return None if no sketch parameters were set.

get_snap()
    Returns the snap setting which may be:
    • True: snap vertices to the nearest pixel center
    • False: leave vertices as-is
    • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel
        center
    Only supported by the Agg and MacOSX backends.

get_transform()
    Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining
affine part of its transformation.

**get_transforms()**

**get_url()**
- Returns the url

**get_urls()**

**get_visible()**
- Return the artist’s visiblity

**get_window_extent(renderer)**

**get_zorder()**
- Return the Artist’s zorder.

**have_units()**
- Return True if units are set on the x or y axes

**hitlist(event)**
- List the children of the artist which contain the mouse event event.

**is_figure_set()**
- Returns True if the artist is assigned to a Figure.

**is_transform_set()**
- Returns True if Artist has a transform explicitly set.

**pchanged()**
- Fire an event when property changed, calling all of the registered callbacks.

**pick(mouseevent)**
- call signature:

  ```python
  pick(mouseevent)
  ```

  each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

**pickable()**
- Return True if Artist is pickable.

**properties()**
- return a dictionary mapping property name -> value for all Artist props

**remove()**
- Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

  Note: relim() will not see collections even if the collection was added to axes with autolim = True.
Note: there is no support for removing the artist’s legend entry.

**remove_callback**(oid)
Remove a callback based on its id.

**See also:**

**add_callback()** For adding callbacks

**set**(**kwargs**)
A tkstyle set command, pass **kwargs** to set properties

**set_agg_filter**(filter_func)
set agg_filter function.

**set_alpha**(alpha)
Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

**set_animated**(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

**set_antialiased**(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds**(aa)
alias for set_antialiased

**set_array**(A)
Set the image array from numpy array A

**set_axes**(axes)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

**set_clim**(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

**set_clip_box**(clipbox)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

**set_clip_on**(b)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]
**set_clip_path**(*path*, *transform=None*)

Set the artist’s clip path, which may be:

- a `Patch` (or subclass) instance
- a `Path` instance, in which case an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- `None`, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

ACCEPTS: `[(Path, Transform), Patch, None]`

**set_cmap**(*cmap*)

set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

**set_color**(*c*)

Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

**set_colorbar**(*args, **kwargs*)

Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

**set_contains**(*picker*)

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return `hit = True` and `props` is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**set_dashes**(*ls*)

alias for `set_linestyle`

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection. `c` can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples
**set_edgecolors**($c$)

alias for set_edgecolor

**set_facecolor**($c$)

Set the facecolor(s) of the collection. $c$ can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_facecolors**($c$)

alias for set_facecolor

**set_figure**(fig)

Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid**(gid)

Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch**(hatch)

Set the hatching pattern

$hatch$ can be one of:

```
/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
O  - large circle
.  - dots
*  - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: [ '/\|+-x.o.\' ]

**set_label**(s)

Set the label to $s$ for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.
**set_linestyle**(*ls*)
Set the linestyle(s) for the collection.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted'] | (offset, on-off-dash-seq)

**set_linestyles**(*ls*)
alias for set_linestyle

**set_linewidth**(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

**set_linewidths**(*lw*)
alias for set_linewidth

**set_lod**(*on*)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

**set_lw**(*lw*)
alias for set_linewidth

**set_norm**(*norm*)
set the normalization instance

**set_offset_position**(*offset_position*)
Set how offsets are applied. If *offset_position* is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If *offset_position* is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

**set_paths**()

**set_picker**(*picker*)
Set the epsilon for picking used by this artist

*picker* can be one of the following:
- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some
artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. if the mouse event is over the artist, return \( \text{hit=}\text{True} \) and \( \text{props} \) is a dictionary of properties you want added to the PickEvent attributes.

  ACCEPTS: [None|float|boolean|callable]

  `set_pickradius(pr)`

  `set_rasterized(rasterized)`
  
  Force rasterized (bitmap) drawing in vector backend output.

  Defaults to None, which implies the backend’s default behavior

  ACCEPTS: [True | False | None]

  `set_sizes(sizes, dpi=72.0)`
  
  Set the sizes of each member of the collection.

  **Parameters**
  
  `sizes`: ndarray or None
  
  The size to set for each element of the collection. The value is the ‘area’ of the element.

  `dpi`: float
  
  The dpi of the canvas. Defaults to 72.0.

  `set_sketch_params(scale=None, length=None, randomness=None)`
  
  Sets the sketch parameters.

  **Parameters**
  
  `scale`: float, optional
  
  The amplitude of the wiggle perpendicular to the source line, in pixels. If `scale` is `None`, or not provided, no sketch filter will be provided.

  `length`: float, optional
  
  The length of the wiggle along the line, in pixels (default 128.0)

  `randomness`: float, optional
  
  The scale factor by which the length is shrunken or expanded (default 16.0)

  `set_snap(snaps)`
  
  Sets the snap setting which may be:
  
  - True: snap vertices to the nearest pixel center
  - False: leave vertices as-is
  - None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

  Only supported by the Agg and MacOSX backends.

  `set_transform(t)`
  
  Set the Transform instance used by this artist.
ACCEPTS: Transform instance

**set_url(url)**
Sets the url for the artist

ACCEPTS: a url string

**set_urls(urls)**

**set_visible(b)**
Set the artist’s visibility.

ACCEPTS: [True | False]

**set_zorder(level)**
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**to_rgba(x, alpha=None, bytes=False)**
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update(props)**
Update the properties of this Artist from the dictionary prop.

**update_from(other)**
copy properties from other to self

**update_scalarmappable()**
If the scalar mappable array is not none, update colors from scalar data

**zorder = 0**

class matplotlib.collections.StarPolygonCollection(numsides, rotation=0, sizes=(1, ), **kwargs)

Bases: matplotlib.collections.RegularPolyCollection

Draw a collection of regular stars with numsides points.
numsides  the number of sides of the polygon  
rotation  the rotation of the polygon in radians  
sizes  gives the area of the circle circumscribing the regular polygon in points^2  

Valid Collection keyword arguments:  
• edgecolors: None  
• facecolors: None  
• linewidths: None  
• antialiaseds: None  
• offsets: None  
• transOffset: transforms.IdentityTransform()  
• norm: None (optional for matplotlib.cm.ScalarMappable)  
• cmap: None (optional for matplotlib.cm.ScalarMappable)  

offsets and transOffset are used to translate the patch after rendering (default no offsets)  

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.

Example: see examples/dynamic_collection.py for complete example:

```python
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5,  # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
)
```

add_callback(func)  
Adds a callback function that will be called whenever one of the Artist’s properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)  
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = u’Artist’

autoscale()  
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()  
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

changed()  
Call this whenever the mappable is changed to notify all the callbackSM listeners to the
‘changed’ signal

**check_update**(checker)

If mappable has changed since the last check, return True; else return False

**contains**(mouseevent)

Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=it), where every item in it contains the event.

**convert_xunits**(x)

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

**convert_yunits**(y)

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

**draw**(artist, renderer, *args, **kwargs)

**findobj**(match=None, include_self=True)

Find artist objects.

Recursively find all Artist instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature boolean = match(artist) used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

**get_agg_filter**()

Return filter function to be used for agg filter

**get_alpha**()

Return the alpha value used for blending - not supported on all backends

**get_animated**()

Return the artist’s animated state

**get_array**()

Return the array

**get_axes**()

Return the Axes instance the artist resides in, or None

**get_children**()

Return a list of the child Artist’s this :class:`Artist` contains.

**get_clim**()

Return the min, max of the color limits for image scaling

**get_clip_box**()

Return artist clipbox

**get_clip_on**()

Return whether artist uses clipping
get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_numsides()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates.
If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`

Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`

Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`

return True if the artist is to be rasterized

`get_rotation()`

`get_sizes()`

Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.

Returns `sizes` : array

The ‘area’ of each element.

`get_sketch_params()`

Returns the sketch parameters for the artist.

Returns `sketch_params` : tuple or None

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

`get_snap()`

Returns the snap setting which may be:

- `True`: snap vertices to the nearest pixel center
- `False`: leave vertices as-is
- `None`: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

`get_transform()`

Return the `Transform` instance used by this artist.
get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining
    affine part of its transformation.

get_transforms()

get_url()
    Returns the url

get_urls()

get_visible()
    Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
    Return the Artist's zorder.

have_units()
    Return True if units are set on the x or y axes

hitlist(event)
    List the children of the artist which contain the mouse event event.

is_figure_set()
    Returns True if the artist is assigned to a Figure.

is_transform_set()
    Returns True if Artist has a transform explicitly set.

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    call signature:

        pick(mouseevent)

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible until
    the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call
    matplotlib.axes.Axes.relim() to update the axes limits if desired.
Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.

Note: there is no support for removing the artist’s legend entry.

**remove_callback(oid)**

Remove a callback based on its id.

See also:

**add_callback()** For adding callbacks

**set(**kwargs**)**

A tkstyle set command, pass `kwargs` to set properties

**set_agg_filter(filter_func)**

Set agg_filter function.

**set_alpha(alpha)**

Set the alpha tranparencies of the collection. `alpha` must be a float or `None`.

ACCEPTS: float or `None`

**set_animated(b)**

Set the artist’s animation state.

ACCEPTS: `[True | False]`

**set_antialiased(aa)**

Set the antialiasing state for rendering.

ACCEPTS: `Boolean` or sequence of booleans

**set_antialiaseds(aa)**

Alias for `set_antialiased`

**set_array(A)**

Set the image array from numpy array `A`

**set_axes(axes)**

Set the `Axes` instance in which the artist resides, if any.

ACCEPTS: an `Axes` instance

**set_clim(vmin=None, vmax=None)**

Set the norm limits for image scaling; if `vmin` is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support `setp`

ACCEPTS: a length 2 sequence of floats

**set_clip_box(clipbox)**

Set the artist’s clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

**set_clip_on(b)**

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.
**set_clip_path(path, transform=None)**
Set the artist’s clip path, which may be:

- a `Patch` (or subclass) instance
- a `Path` instance, in which case an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- `None`, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

**Accepts:**  
[True | False]

**set_cmap(cmap)**
set the colormap for luminance data

**Accepts:**  
a colormap or registered colormap name

**set_color(c)**
Set both the edgecolor and the facecolor.

**Accepts:**  
matplotlib color arg or sequence of rgba tuples

**See also:**

- `set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

**set_colorbar(*args, **kwargs)**
Deprecated since version 1.3: The `set_colorbar` function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

**set_contains(picker)**
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return `hit = True` and `props` is a dictionary of properties you want returned with the contains test.

**Accepts:**  
a callable function

**set_dashes(ls)**
alias for `set_linestyle`

**set_edgecolor(c)**
Set the edgecolor(s) of the collection. `c` can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

**Accepts:**  
matplotlib color arg or sequence of rgba tuples
set_edgecolors(c)
alias for set_edgecolor

set_facecolor(c)
Set the facecolor(s) of the collection. c can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_facecolors(c)
alias for set_facecolor

set_figure(fig)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

set_gid(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(hatch)
Set the hatching pattern

hatch can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
. - dots
* - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.
**set_linestyle**(ls)
Set the linestyle(s) for the collection.

ACCEPTS: ['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq)]

**set_linestyles**(ls)
alias for set_linestyle

**set_linewidth**(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

**set_linewidths**(lw)
alias for set_linewidth

**set_lod**(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

**set_lw**(lw)
alias for set_linewidth

**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(path_effects)
set path_effects, which should be a list of instances of matplotlib.pathEffect._Base class or its derivatives.

**set_paths**()

**set_picker**(picker)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some
artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

• A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. If the mouse event is over the artist, return `hit=True` and `props` is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius** *(pr)*

**set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sizes**(sizes, dpi=72.0)

Set the sizes of each member of the collection.

**Parameters sizes**: ndarray or None

The size to set for each element of the collection. The value is the ‘area’ of the element.

**dpi**: float

The dpi of the canvas. Defaults to 72.0.

**set_sketch_params**(scale=None, length=None, randomness=None)

Sets the sketch parameters.

**Parameters scale**: float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

**length**: float, optional

The length of the wiggle along the line, in pixels (default 128.0)

**randomness**: float, optional

The scale factor by which the length is shrunk or expanded (default 16.0)

**set_snap**(snap)

Sets the snap setting which may be:

• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)

Set the Transform instance used by this artist.
ACCEPTS: Transform instance

**set_url(url)**
Sets the url for the artist

ACCEPTS: a url string

**set_urls(urls)**

**set_visible(b)**
Set the artist’s visiblity.

ACCEPTS: [True | False]

**set_zorder(level)**
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**to_rgba(x, alpha=None, bytes=False)**
Return a normalized rgba array corresponding to x.

In the normal case, $x$ is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If $x$ is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the $alpha$ kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the $alpha$ kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if $bytes$ is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as $x$ being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update(props)**
Update the properties of this Artist from the dictionary prop.

**update_from(other)**
copy properties from other to self

**update_scalarmappable()**
If the scalar mappable array is not none, update colors from scalar data

**zorder = 0**

class matplotlib.collections.TriMesh(triangulation, **kwargs)
Bases: matplotlib.collections.Collection

Class for the efficient drawing of a triangular mesh using Gouraud shading.
A triangular mesh is a Triangulation object.

**add_callback(func)**

Adds a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

**add_checker(checker)**

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = u’Artist’

**autoscale()**

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None()**

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**changed()**

Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

**check_update(checker)**

If mappable has changed since the last check, return True; else return False

**contains(mouseevent)**

Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

**static convert_mesh_to_paths(tri)**

Converts a given mesh into a sequence of matplotlib.path.Path objects for easier rendering by backends that do not directly support meshes.

This function is primarily of use to backend implementers.

**convert_xunits(x)**

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

**convert_yunits(y)**

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

**draw(artist, renderer, *args, **kwargs)**

**findobj(match=None, include_self=True)**

Find artist objects.

Recursively find all Artist instances contained in self.

match can be

• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.
get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None

get_children()
Return a list of the child Artist’s this :class:‘Artist contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or None for default.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the Figure instance the artist belongs to.
get_gid()
   Returns the group id

get_hatch()
   Return the current hatching pattern

get_label()
   Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
   Returns how offsets are applied for the collection. If offset_position is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
   Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
   Return the picker object used by this artist

get_pickradius()

get_rasterized()
   return True if the artist is to be rasterized

get_sketch_params()
   Returns the sketch parameters for the artist.
      Returns sketch_params : tuple or None
      A 3-tuple with the following elements:
      • scale: The amplitude of the wiggle perpendicular to the source line.
      • length: The length of the wiggle along the line.
randomness: The scale factor by which the length is shrunken or expanded.
May return None if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
Only supported by the Agg and MacOSX backends.

get_transform()
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
Return the Artist’s zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
call signature:
pick(mouseevent)

Each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set.

**pickable()**
Return True if Artist is pickable.

**properties()**
Return a dictionary mapping property name -> value for all Artist props.

**remove()**
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

**remove_callback(oid)**
Remove a callback based on its id.

See also:
**add_callback()** For adding callbacks

**set(**kwargs**)**
A tkstyle set command, pass kwargs to set properties

**set_agg_filter(filter_func)**
set agg_filter function.

**set_alpha(alpha)**
Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

**set_animated(b)**
Set the artist’s animation state.

ACCEPTS: [True | False]

**set_antialiased(aa)**
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds(aa)**
alias for set_antialiased

**set_array(A)**
Set the image array from numpy array A

**set_axes(ax)**
Set the Axes instance in which the artist resides, if any.
ACCEPTEs: an Axes instance

**set_clim**(vmin=None, vmax=None)

set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTEs: a length 2 sequence of floats

**set_clip_box**(clipbox)

Set the artist’s clip Bbox.

ACCEPTEs: a matplotlib.transforms.Bbox instance

**set_clip_on**(b)

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTEs: [True | False]

**set_clip_path**(path, transform=None)

Set the artist’s clip path, which may be:

- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTEs: [(Path, Transform) | Patch | None ]

**set_cmap**(cmap)

set the colormap for luminance data

ACCEPTEs: a colormap or registered colormap name

**set_color**(c)

Set both the edgecolor and the facecolor.

ACCEPTEs: matplotlib color arg or sequence of rgba tuples

See also:

**set_facecolor()**, **set_edgecolor()** For setting the edge or face color individually.

**set_colorbar**(args, **kwargs)

Deprecated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set the colorbar and axes instances associated with mappable

**set_contains**(picker)

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)
If the mouse event is over the artist, return \textit{hit} = \textit{True} and \textit{props} is a dictionary of properties you want returned with the contains test.

\textbf{ACCEPTS}: a callable function

\textbf{set\_dashes}(ls)

\textit{ls} is the line style

\textbf{set\_edgecolor}(c)

Set the edgecolor(s) of the collection. \textit{c} can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If \textit{c} is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

\textbf{ACCEPTS}: matplotlib color arg or sequence of rgba tuples

\textbf{set\_edgecolors}(c)

\textbf{set\_facecolor}(c)

Set the facecolor(s) of the collection. \textit{c} can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If \textit{c} is ‘none’, the patch will not be filled.

\textbf{ACCEPTS}: matplotlib color arg or sequence of rgba tuples

\textbf{set\_facecolors}(c)

\textbf{set\_figure}(fig)

Set the \texttt{Figure} instance the artist belongs to.

\textbf{ACCEPTS}: a \texttt{matplotlib.figure.Figure} instance

\textbf{set\_gid}(gid)

Sets the (group) id for the artist

\textbf{ACCEPTS}: an id string

\textbf{set\_hatch}(hatch)

Set the hatching pattern

\textit{hatch} can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
0 - small circle
0 - large circle
Matplotlib, Release 1.4.3

. - dots
* - stars

Letters can be combined, in which case all the specified hatchings are done. If same letter
repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the
collection as a whole, not separately for each member.

ACCEPTS: [‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

set_linestyle(ls)
Set the linestyle(s) for the collection.

ACCEPTS: [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) ]

set_linestyles(ls)
alias for set_linestyle

set_linewidth(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the
patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(lw)
alias for set_linewidth

set_lod(on)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the
axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

set_lw(lw)
alias for set_linewidth

set_norm(norm)
set the normalization instance

set_offset_position(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after
the master transform has been applied, that is, the offsets are in screen coordinates. If
offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in
data coordinates.

set_offsets(offssets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats
**set_path_effects**(path_effects)

set path_effects, which should be a list of instances of matplotlib.path.effect._Base class or its derivatives.

**set_paths()**

**set_picker**(picker)

Set the epsilon for picking used by this artist

picker can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
def picker(artist, mouseevent):
    hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sketch_params**(scale=None, length=None, randomness=None)

Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform(t)**
Set the Transform instance used by this artist.

**ACCEPTS:** Transform instance

**set_url(url)**
Sets the url for the artist

**ACCEPTS:** a url string

**set_urls(urls)**

**set_visible(b)**
Set the artist’s visibility.

**ACCEPTS:** [True | False]

**set_zorder(level)**
Set the zorder for the artist. Artists with lower zorder values are drawn first.

**ACCEPTS:** any number

**to_rgba(x, alpha=None, bytes=False)**
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update(props)**
Update the properties of this Artist from the dictionary prop.

**update_from(other)**
copy properties from other to self

**update_scalarmappable()**
If the scalar mappable array is not none, update colors from scalar data
zorder = 0
45.1 `matplotlib.colorbar`

Colorbar toolkit with two classes and a function:

- **ColorbarBase** the base class with full colorbar drawing functionality. It can be used as-is to make a colorbar for a given colormap; a mappable object (e.g., image) is not needed.
- **Colorbar** the derived class for use with images or contour plots.
- **make_axes()** a function for resizing an axes and adding a second axes suitable for a colorbar

The `colorbar()` method uses `make_axes()` and `Colorbar`; the `colorbar()` function is a thin wrapper over `colorbar()`.

```python
class matplotlib.colorbar.Colorbar(ax, mappable, **kw)
```

This class connects a `ColorbarBase` to a `ScalarMappable` such as a `AxesImage` generated via `imshow()`.

It is not intended to be instantiated directly; instead, use `colorbar()` or `colorbar()` to make your colorbar.

- **add_lines(CS, erase=True)**
  Add the lines from a non-filled `ContourSet` to the colorbar.
  Set `erase` to False if these lines should be added to any pre-existing lines.

- **on_mappable_changed(mappable)**
  Updates this colorbar to match the mappable’s properties.
  Typically this is automatically registered as an event handler by `colorbar_factory()` and should not be called manually.

- **remove()**
  Remove this colorbar from the figure. If the colorbar was created with `use_gridspec=True` then restore the gridspec to its previous value.

- **update_bruteforce(mappable)**
  Destroy and rebuild the colorbar. This is intended to become obsolete, and will probably
be deprecated and then removed. It is not called when the pyplot.colorbar function or the Figure.colorbar method are used to create the colorbar.

```python
update_normal(mappable)
```

update solid, lines, etc. Unlike update_bruteforce, it does not clear the axes. This is meant to be called when the image or contour plot to which this colorbar belongs is changed.

class matplotlib.colorbar.ColorbarBase(ax, cmap=None, norm=None, alpha=None, values=None, boundaries=None, orientation='vertical', ticklocation='auto', extend='neither', spacing='uniform', ticks=None, format=None, drawedges=False, filled=True, extendfrac=None, extendrect=False, label=u'')

Bases: matplotlib.cm.ScalarMappable

Draw a colorbar in an existing axes.

This is a base class for the Colorbar class, which is the basis for the colorbar() function and the colorbar() method, which are the usual ways of creating a colorbar.

It is also useful by itself for showing a colormap. If the cmap kwarg is given but boundaries and values are left as None, then the colormap will be displayed on a 0-1 scale. To show the under- and over-value colors, specify the norm as:

```python
colors.Normalize(clip=False)
```

To show the colors versus index instead of on the 0-1 scale, use:

```python
norm=colors.NoNorm.
```

Useful attributes:
- `ax` the Axes instance in which the colorbar is drawn
- `lines` a list of LineCollection if lines were drawn, otherwise an empty list
- `dividers` a LineCollection if drawedges is True, otherwise None

Useful public methods are set_label() and add_lines().

```python
add_lines(levels, colors, linewidths, erase=True)
```

Draw lines on the colorbar.

`colors` and `linewidths` must be scalars or sequences the same length as `levels`.

Set `erase` to False to add lines without first removing any previously added lines.

```python
ax = None
```

The axes that this colorbar lives in.

```python
config_axis()
```

```python
draw_all()
```

Calculate any free parameters based on the current cmap and norm, and do all the drawing.

```python
remove()
```

Remove this colorbar from the figure.
**set_alpha(alpha)**

**set_label(label, **kw)**
Label the long axis of the colorbar

**set_ticklabels(ticklabels, update_ticks=True)**
set tick labels. Tick labels are updated immediately unless update_ticks is False. To manually update the ticks, call update_ticks method explicitly.

**set_ticks(ticks, update_ticks=True)**
set tick locations. Tick locations are updated immediately unless update_ticks is False. To manually update the ticks, call update_ticks method explicitly.

**update_ticks()**
Force the update of the ticks and ticklabels. This must be called whenever the tick locator and/or tick formatter changes.

**class matplotlib.colorbar.ColorbarPatch(ax, mappable, **kw)**
Bases: matplotlib.colorbar.Colorbar
A Colorbar which is created using Patch rather than the default pcolor(). It uses a list of Patch instances instead of a PatchCollection because the latter does not allow the hatch pattern to vary among the members of the collection.

**matplotlib.colorbar.colorbar_factory(cax, mappable, **kwargs)**
Creates a colorbar on the given axes for the given mappable. Typically, for automatic colorbar placement given only a mappable use colorbar().

**matplotlib.colorbar.make_axes(parents, location=None, orientation=None, fraction=0.15, shrink=1.0, aspect=20, **kw)**
Resize and reposition parent axes, and return a child axes suitable for a colorbar:

cax, kw = make_axes(parent, **kw)

Keyword arguments may include the following (with defaults):

- **location** (None or 'left', 'right', 'top', 'bottom') The position, relative to parents, where the colorbar axes should be created. If None, the value will either come from the given orientation, else it will default to ‘right’.
- **orientation** (None or 'vertical', 'horizontal') The orientation of the colorbar. Typically, this keyword shouldn’t be used, as it can be derived from the location keyword.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>panchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.

**matplotlib.colorbar.make_axes_gridspec**(parent, **kw**)

Resize and reposition a parent axes, and return a child axes suitable for a colorbar. This function is similar to make_axes. Primary differences are

- *make_axes_gridspec* only handles the orientation keyword and cannot handle the “location” keyword.
- *make_axes_gridspec* should only be used with a subplot parent.
- *make_axes creates an instance of Axes. make_axes_gridspec* creates an instance of Subplot.
- *make_axes updates the position of the* parent. *make_axes_gridspec* replaces the grid_spec attribute of the parent with a new one.

While this function is meant to be compatible with *make_axes*, there could be some minor differences:

cax, kw = make_axes_gridspec(parent, **kw)

Keyword arguments may include the following (with defaults):

- **orientation** ‘vertical’ or ‘horizontal’

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>panchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

All but the first of these are stripped from the input kw set.
Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.
COLORS

For a visual representation of the matplotlib colormaps, see the “Color” section in the gallery.

46.1 matplotlib.colors

A module for converting numbers or color arguments to *RGB* or *RGBA*

*RGB* and *RGBA* are sequences of, respectively, 3 or 4 floats in the range 0-1.

This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap. Colormapping typically involves two steps: a data array is first mapped onto the range 0-1 using an instance of `Normalize` or of a subclass; then this number in the 0-1 range is mapped to a color using an instance of a subclass of `Colormap`. Two are provided here: `LinearSegmentedColormap`, which is used to generate all the built-in colormap instances, but is also useful for making custom colormaps, and `ListedColormap`, which is used for generating a custom colormap from a list of color specifications.

The module also provides a single instance, `colorConverter`, of the `ColorConverter` class providing methods for converting single color specifications or sequences of them to *RGB* or *RGBA*.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

- b: blue
- g: green
- r: red
- c: cyan
- m: magenta
- y: yellow
- k: black
- w: white

Gray shades can be given as a string encoding a float in the 0-1 range, e.g.:
color = '0.75'

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

color = '#eeeeff'

or you can pass an $R, G, B$ tuple, where each of $R, G, B$ are in the range [0,1].

Finally, legal html names for colors, like ‘red’, ‘burlywood’ and ‘chartreuse’ are supported.

class matplotlib.colors.BoundaryNorm(boundaries, ncolors, clip=False)

Bases: matplotlib.colors.Normalize

Generate a colormap index based on discrete intervals.

Unlike Normalize or LogNorm, BoundaryNorm maps values to integers instead of to the interval 0-1.

Mapping to the 0-1 interval could have been done via piece-wise linear interpolation, but using integers seems simpler, and reduces the number of conversions back and forth between integer and floating point.

*boundaries* a monotonically increasing sequence

*ncolors* number of colors in the colormap to be used

If:

$$b[i] <= v < b[i+1]$$

then $v$ is mapped to color $j$; as $i$ varies from 0 to len(boundaries)-2, $j$ goes from 0 to ncolors-1.

Out-of-range values are mapped to -1 if low and ncolors if high; these are converted to valid indices by Colormap.__call__().

inverse(value)

class matplotlib.colors.ColorConverter

Bases: object

Provides methods for converting color specifications to RGB or RGBA

Caching is used for more efficient conversion upon repeated calls with the same argument.

Ordinarily only the single instance instantiated in this module, colorConverter, is needed.

cache = {'green': (0.0, 0.5019607843137255, 0.0), '0.8': (0.8, 0.8, 0.8), '#ffed6f': (1.0, 0.9294117647058824, 0.43529411764705883), '#467821': (0.27450980392156865, 0.47058823529411764, 0.13333333333333333), 'chartreuse': (0.5058823529411764, 0.4549019607843137, 0.0), '#b3de69': (0.7019607843137254, 0.8705882352941177, 0.4117647058823529), 'firebrick': (0.6980392156862745, 0.13333333333333333, 0.13333333333333333), 'burlywood': (0.7372549019607844, 0.7372549019607844, 0.7372549019607844), 'red': (1.0, 0.0, 0.0), 'orange': (0.75, 0.75, 0), 'black': (0.0, 0.0, 0.0), 'white': (1.0, 1.0, 1.0), 'gray': (0.5, 0.5, 0.5), 'blue': (0.0, 0.5, 0.0), 'cyan': (0.0, 1.0, 1.0), 'magenta': (1.0, 0.0, 1.0), 'yellow': (1.0, 0.75, 0.75), 'lightgreen': (0.0, 1.0, 0.5), 'lightblue': (0, 0.5, 0.8), 'lightyellow': (0, 1.0, 0.5), 'lightcyan': (0, 0.5, 0.8), 'lightmagenta': (0, 0.5, 0.8), 'lightgray': (0.5, 0.5, 0.5), 'lightblack': (0.0, 0.0, 0.0), 'lightwhite': (1.0, 1.0, 1.0), 'lightred': (1.0, 0.0, 0.0), 'lightyellow': (1.0, 0.75, 0.75) }

colors = {l: c for l, c in colors.items()}

to_rgb(arg)

Returns an RGB tuple of three floats from 0-1.

arg can be an RGB or RGBA sequence or a string in any of several forms:

1. a letter from the set ‘rgbcmykw’
2. a hex color string, like ‘#00FFFF’
3. a standard name, like ‘aqua’
4. a string representation of a float, like ‘0.4’, indicating gray on a 0-1 scale
   if \( \text{arg} \) is RGBA, the \( A \) will simply be discarded.

**to_rgba** (arg, alpha=None)

Returns an RGBA tuple of four floats from 0-1.

For acceptable values of \( \text{arg} \), see **to_rgb()**. In addition, if \( \text{arg} \) is “none” (case-insensitive), then \((0,0,0,0)\) will be returned. If \( \text{arg} \) is an RGBA sequence and \( \text{alpha} \) is not None, \( \text{alpha} \) will replace the original \( A \).

**to_rgba_array** (c, alpha=None)

Returns a numpy array of RGBA tuples.

Accepts a single mpl color spec or a sequence of specs.

Special case to handle “no color”: if \( \text{c} \) is “none” (case-insensitive), then an empty array will be returned. Same for an empty list.

**class** matplotlib.colors.Colormap(name, \( N=256 \))

Bases: object

Baseclass for all scalar to RGBA mappings.

Typically Colormap instances are used to convert data values (floats) from the interval \([0, 1]\) to the RGBA color that the respective Colormap represents. For scaling of data into the \([0, 1]\) interval see **matplotlib.colors.Normalize**. It is worth noting that matplotlib.cm.ScalarMappable subclasses make heavy use of this data->normalize->map-to-color processing chain.

**Parameters**

- **name** : str
  The name of the colormap.
- **N** : int
  The number of rgb quantization levels.

**colorbar_extend** = None

When this colormap exists on a scalar mappable and colorbar_extend is not False, colorbar creation will pick up **colorbar_extend** as the default value for the extend keyword in the matplotlib.colorbar.Colorbar constructor.

**is_gray**()

**set_bad** (color=u’k’, alpha=None)

Set color to be used for masked values.

**set_over** (color=u’k’, alpha=None)

Set color to be used for high out-of-range values. Requires \( \text{norm.clip} = \text{False} \)

**set_under** (color=u’k’, alpha=None)

Set color to be used for low out-of-range values. Requires \( \text{norm.clip} = \text{False} \)

**class** matplotlib.colors.LightSource(azdeg=315, altdeg=45, hsv_min_val=0, hsv_max_val=1, hsv_min_sat=1, hsv_max_sat=0)

Bases: object
Create a light source coming from the specified azimuth and elevation. Angles are in degrees, with the azimuth measured clockwise from north and elevation up from the zero plane of the surface. The `shade()` is used to produce rgb values for a shaded relief image given a data array.

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

The color of the resulting image will be darkened by moving the (s,v) values (in hsv colorspace) toward (hsv_min_sat, hsv_min_val) in the shaded regions, or lightened by sliding (s,v) toward (hsv_max_sat, hsv_max_val) in regions that are illuminated. The default extremes are chose so that completely shaded points are nearly black (s = 1, v = 0) and completely illuminated points are nearly white (s = 0, v = 1).

```python
shade(data, cmap, norm=None)
```

Take the input data array, convert to HSV values in the given colormap, then adjust those color values to give the impression of a shaded relief map with a specified light source. RGBA values are returned, which can then be used to plot the shaded image with imshow.

```python
shade_rgb(rgb, elevation, fraction=1.0)
```

Take the input RGB array (ny*nx*3) adjust their color values to given the impression of a shaded relief map with a specified light source using the elevation (ny*nx). A new RGB array ((ny*nx*3)) is returned.

```python
class matplotlib.colors.LinearSegmentedColormap(name, segmentdata, N=256, gamma=1.0)
```

Colormap objects based on lookup tables using linear segments.

The lookup table is generated using linear interpolation for each primary color, with the 0-1 domain divided into any number of segments.

Create color map from linear mapping segments

segmentdata argument is a dictionary with a red, green and blue entries. Each entry should be a list of x, y0, y1 tuples, forming rows in a table. Entries for alpha are optional.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

```python
cdict = {'red': [(0.0, 0.0, 0.0),
                 (0.5, 1.0, 1.0),
                 (1.0, 1.0, 1.0)],
          'green': [(0.0, 0.0, 0.0),
                     (0.25, 0.0, 0.0),
                     (0.75, 1.0, 1.0),
                     (1.0, 1.0, 1.0)],
          'blue': [(0.0, 0.0, 0.0),
                   (0.5, 0.0, 0.0),
                   (1.0, 1.0, 1.0)]}
```
Each row in the table for a given color is a sequence of \( x, y0, y1 \) tuples. In each sequence, \( x \) must increase monotonically from 0 to 1. For any input value \( z \) falling between \( x[i] \) and \( x[i+1] \), the output value of a given color will be linearly interpolated between \( y1[i] \) and \( y0[i+1] \):

\[
\text{row i: } x \quad y0 \quad y1 \\
\text{row i+1: } x \quad y0 \quad y1
\]

Hence \( y0 \) in the first row and \( y1 \) in the last row are never used.

See also:

LinearSegmentedColormap.from_list() Static method; factory function for generating a smoothly-varying LinearSegmentedColormap.

makeMappingArray() For information about making a mapping array.

static from_list(name, colors, \( N=256 \), gamma=1.0)
Make a linear segmented colormap with name from a sequence of colors which evenly transitions from colors[0] at \( \text{val}=0 \) to colors[-1] at \( \text{val}=1 \). \( N \) is the number of rgb quantization levels. Alternatively, a list of (value, color) tuples can be given to divide the range unevenly.

set_gamma(gamma)
Set a new gamma value and regenerate color map.

class matplotlib.colors.ListedColormap(colors, name=u'from_list', \( N=None \))
Colormap object generated from a list of colors.

This may be most useful when indexing directly into a colormap, but it can also be used to generate special colormaps for ordinary mapping.

Make a colormap from a list of colors.

\( \text{colors} \) a list of matplotlib color specifications, or an equivalent Nx3 or Nx4 floating point array (\( N \) rgb or rgba values)

\( \text{name} \) a string to identify the colormap

\( N \) the number of entries in the map. The default is None, in which case there is one colormap entry for each element in the list of colors. If:

\[
N < \text{len(colors)}
\]
the list will be truncated at \( N \). If:

\[
N > \text{len(colors)}
\]
the list will be extended by repetition.

class matplotlib.colors.LogNorm(vmin=None, vmax=None, clip=False)

Normalize a given value to the 0-1 range on a log scale
If \( v_{\text{min}} \) or \( v_{\text{max}} \) is not given, they are taken from the input’s minimum and maximum value respectively. If \( \text{clip} \) is \( True \) and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

\[ v_{\text{min}} == v_{\text{max}} \]

Works with scalars or arrays, including masked arrays. If \( \text{clip} \) is \( True \), masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is \( \text{clip} = False \).

\textbf{autoscale}(A)

Set \( v_{\text{min}} \), \( v_{\text{max}} \) to min, max of \( A \).

\textbf{autoscale_None}(A)

autoscale only None-valued \( v_{\text{min}} \) or \( v_{\text{max}} \)

\textbf{inverse}(value)

class matplotlib.colors.NoNorm\( (v_{\text{min}}=\text{None}, v_{\text{max}}=\text{None}, \text{clip}=\text{False}) \)

Bases: matplotlib.colors.Normalize

Dummy replacement for Normalize, for the case where we want to use indices directly in a ScalarMappable.

If \( v_{\text{min}} \) or \( v_{\text{max}} \) is not given, they are taken from the input’s minimum and maximum value respectively. If \( \text{clip} \) is \( True \) and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

\[ v_{\text{min}} == v_{\text{max}} \]

Works with scalars or arrays, including masked arrays. If \( \text{clip} \) is \( True \), masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is \( \text{clip} = False \).

\textbf{inverse}(value)

class matplotlib.colors.Normalize\( (v_{\text{min}}=\text{None}, v_{\text{max}}=\text{None}, \text{clip}=\text{False}) \)

Bases: object

A class which, when called, can normalize data into the \([0.0, 1.0]\) interval.

If \( v_{\text{min}} \) or \( v_{\text{max}} \) is not given, they are taken from the input’s minimum and maximum value respectively. If \( \text{clip} \) is \( True \) and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

\[ v_{\text{min}} == v_{\text{max}} \]

Works with scalars or arrays, including masked arrays. If \( \text{clip} \) is \( True \), masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and
masked colors in the colormap, so it is likely to lead to surprises; therefore the default is $clip = False$.

```python
autoscale(A)
    Set $vmin, vmax$ to min, max of $A$.
```

```python
autoscale_None(A)
    autoscale only None-valued vmin or vmax
```

```python
inverse(value)
```

```python
static process_value(value)
    Homogenize the input $value$ for easy and efficient normalization.

    $value$ can be a scalar or sequence.

    Returns $result, is_scalar$, where $result$ is a masked array matching $value$. Float dtypes are preserved; integer types with two bytes or smaller are converted to np.float32, and larger types are converted to np.float. Preserving float32 when possible, and using in-place operations, can greatly improve speed for large arrays.

    Experimental; we may want to add an option to force the use of float32.

```python
scaled()
    return true if vmin and vmax set
```

```python
class matplotlib.colors.PowerNorm(gamma, vmin=None, vmax=None, clip=False)
    Bases: matplotlib.colors.Normalize
```

Normalizes a given value to the $[0, 1]$ interval with a power-law scaling. This will clip any negative data points to 0.

```python
autoscale(A)
    Set $vmin, vmax$ to min, max of $A$.
```

```python
autoscale_None(A)
    autoscale only None-valued vmin or vmax
```

```python
inverse(value)
```

```python
class matplotlib.colors.SymLogNorm(linthresh, linscale=1.0, vmin=None, vmax=None, clip=False)
    Bases: matplotlib.colors.Normalize
```

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter $linthresh$ allows the user to specify the size of this range ($-linthresh, linthresh$).

$linthresh$: The range within which the plot is linear (to avoid having the plot go to infinity around zero).

$linscale$: This allows the linear range ($-linthresh$ to $linthresh$) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when
linyscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range. Defaults to 1.

**autoscale** *(A)*

Set vmin, vmax to min, max of A.

**autoscale_None** *(A)*

autoscale only None-valued vmin or vmax

**inverse** *(value)*

```
matplotlib.colors.from_levels_and_colors(levels, colors, extend='neither')
```

A helper routine to generate a cmap and a norm instance which behave similar to contour’s levels and colors arguments.

**Parameters**

- `levels`: sequence of numbers
  - The quantization levels used to construct the BoundaryNorm. Values v are quantized to level i if `lev[i] <= v < lev[i+1]`.

- `colors`: sequence of colors
  - The fill color to use for each level. If `extend` is “neither” there must be `n_level - 1` colors. For an `extend` of “min” or “max” add one extra color, and for an `extend` of “both” add two colors.

- `extend`: {'neither', 'min', 'max', 'both'}, optional
  - The behaviour when a value falls out of range of the given levels. See `contourf()` for details.

**Returns** *(cmap, norm)*

- `cmap`: `Colormap`
- `norm`: `Normalize` instance

```
matplotlib.colors.hex2color(s)
```

Take a hex string `s` and return the corresponding rgb 3-tuple Example: #efefef -> (0.93725, 0.93725, 0.93725)

```
matplotlib.colors.hsv_to_rgb(hsv)
```

convert hsv values in a numpy array to rgb values all values assumed to be in range [0, 1]

**Parameters**

- `hsv`: (..., 3) array-like
  - All values assumed to be in range [0, 1]

**Returns**

- `rgb`: (..., 3) ndarray
  - Colors converted to RGB values in range [0, 1]

```
matplotlib.colors.is_color_like(c)
```

Return `True` if `c` can be converted to `RGB`

```
matplotlib.colors.makeMappingArray(N, data, gamma=1.0)
```

Create an `N`-element 1-d lookup table

**data** represented by a list of x, y0, y1 mapping correspondences. Each element in this list represents how a value between 0 and 1 (inclusive) represented by x is mapped to a corresponding value between 0 and 1 (inclusive). The two values of y are to allow for discontinuous mapping functions (say as might be found in a sawtooth) where y0 represents the value of y for values of x <= to that given, and y1 is the value to be used for x > than that given). The list must start with x=0, end with x=1, and all values of x must be in increasing order. Values between the given mapping points are determined by simple linear interpolation.

Alternatively, data can be a function mapping values between 0 - 1 to 0 - 1.
The function returns an array “result” where \( \text{result}[x \times (N-1)] \) gives the closest value for values of \( x \) between 0 and 1.

\( \text{matplotlib.colors.no_norm}(*\text{args}, **\text{kwargs}) \)

Deprecated since version 1.3: The no_norm class alias was deprecated in version 1.3. Use NoNorm instead.

Dummy replacement for Normalize, for the case where we want to use indices directly in a ScalarMappable.

\( \text{matplotlib.colors.normalize}(*\text{args}, **\text{kwargs}) \)

Deprecated since version 1.3: The normalize class alias was deprecated in version 1.3. Use Normalize instead.

A class which, when called, can normalize data into the \([0.0, 1.0]\) interval.

\( \text{matplotlib.colors.rgb2hex}(\text{rgb}) \)

Given an rgb or rgba sequence of 0-1 floats, return the hex string

\( \text{matplotlib.colors.rgb_to_hsv}(\text{arr}) \)

convert float rgb values (in the range \([0, 1]\)), in a numpy array to hsv values.

Parameters

\( \text{arr} : (..., 3) \) array-like

All values must be in the range \([0, 1]\)

Returns

\( \text{hsv} : (..., 3) \) ndarray

Colors converted to hsv values in range \([0, 1]\)
47.1 matplotlib.dates

Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of Python datetime, the add-on modules pytz and dateutil. datetime objects are converted to floating point numbers which represent time in days since 0001-01-01 UTC, plus 1. For example, 0001-01-01, 06:00 is 1.25, not 0.25. The helper functions date2num(), num2date() and drange() are used to facilitate easy conversion to and from datetime and numeric ranges.

**Note:** Like Python’s datetime, mpl uses the Gregorian calendar for all conversions between dates and floating point numbers. This practice is not universal, and calendar differences can cause confusing differences between what Python and mpl give as the number of days since 0001-01-01 and what other software and databases yield. For example, the US Naval Observatory uses a calendar that switches from Julian to Gregorian in October, 1582. Hence, using their calculator, the number of days between 0001-01-01 and 2006-04-01 is 732403, whereas using the Gregorian calendar via the datetime module we find:

In [31]: date(2006,4,1).toordinal() - date(1,1,1).toordinal()
Out[31]: 732401
A wide range of specific and general purpose date tick locators and formatters are provided in this module. See `matplotlib.ticker` for general information on tick locators and formatters. These are described below.

All the matplotlib date converters, tickers and formatters are timezone aware, and the default timezone is given by the timezone parameter in your `matplotlibrc` file. If you leave out a `tz` timezone instance, the default from your rc file will be assumed. If you want to use a custom time zone, pass a `pytz.timezone` instance with the `tz` keyword argument to `num2date()`, `plot_date()`, and any custom date tickers or locators you create. See `pytz` for information on `pytz` and timezone handling.

The `dateutil` module provides additional code to handle date ticking, making it easy to place ticks on any kinds of dates. See examples below.

### 47.1.1 Date tickers

Most of the date tickers can locate single or multiple values. For example:

```python
# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU

# tick on mondays every week
loc = WeekdayLocator(byweekday=MO, tz=tz)

# tick on mondays and saturdays
loc = WeekdayLocator(byweekday=(MO, SA))
```

In addition, most of the constructors take an interval argument:

```python
# tick on mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The `rrule` locator allows completely general date ticking:

```python
# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
```

Here are all the date tickers:

- `MinuteLocator`: locate minutes
- `HourLocator`: locate hours
- `DayLocator`: locate specified days of the month
- `WeekdayLocator`: locate days of the week, e.g., MO, TU
- `MonthLocator`: locate months, e.g., 7 for july
- `YearLocator`: locate years that are multiples of base
• **RRuleLocator**: locate using a `matplotlib.dates.rrulewrapper`. The `rrulewrapper` is a simple wrapper around a `dateutil.rrule` (dateutil) which allow almost arbitrary date tick specifications. See rrule example.

• **AutoDateLocator**: On autoscale, this class picks the best `MultipleDateLocator` to set the view limits and the tick locations.

### 47.1.2 Date formatters

Here all all the date formatters:

• **AutoDateFormatter**: attempts to figure out the best format to use. This is most useful when used with the `AutoDateLocator`.

• **DateFormatter**: use `strftime()` format strings

• **IndexDateFormatter**: date plots with implicit `x` indexing.

`matplotlib.dates.date2num(d)`

`d` is either a `datetime` instance or a sequence of datetimes.

Return value is a floating point number (or sequence of floats) which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC, plus one. The addition of one here is a historical artifact. Also, note that the Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.

`matplotlib.dates.num2date(x, tz=None)`

`x` is a float value which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC plus one. The addition of one here is a historical artifact. Also, note that the Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.

Return value is a `datetime` instance in timezone `tz` (default to rcparams TZ value).

If `x` is a sequence, a sequence of `datetime` objects will be returned.

`matplotlib.dates.drange(dstart, dend, delta)`

Return a date range as float Gregorian ordinals. `dstart` and `dend` are `datetime` instances. `delta` is a `datetime.timedelta` instance.

`matplotlib.dates.epoch2num(e)`

Convert an epoch or sequence of epochs to the new date format, that is days since 0001.

`matplotlib.dates.num2epoch(d)`

Convert days since 0001 to epoch. `d` can be a number or sequence.

`matplotlib.dates.mx2num(mxdates)`

Convert mx `datetime` instance (or sequence of mx instances) to the new date format.

`class matplotlib.dates.DateFormatter(fmt, tz=None)`

Bases: `matplotlib.ticker.Formatter`

Tick location is seconds since the epoch. Use a `strftime()` format string.
Python only supports `datetime.strftime()` formatting for years greater than 1900. Thanks to Andrew Dalke, Dalke Scientific Software who contributed the `strftime()` code below to include dates earlier than this year.

`fmt` is an `strftime()` format string; `tz` is the tzinfo instance.

```python
illegal_s = <_sre.SRE_Pattern object at 0x7f8bf7ef1a80>
```

```python
set_tzinfo(tz)
```

```python
strftime(dt, fmt)
```

```python
class matplotlib.dates.IndexDateFormatter(t, fmt, tz=None):
    Bases: matplotlib.ticker.Formatter

    Use with IndexLocator to cycle format strings by index.

    `t` is a sequence of dates (floating point days). `fmt` is a `strftime()` format string.

```python
class matplotlib.dates.AutoDateFormatter(locator, tz=None, defaultfmt=u'%Y-%m-%d'):
    Bases: matplotlib.ticker.Formatter

    This class attempts to figure out the best format to use. This is most useful when used with the
    AutoDateLocator.

    The AutoDateFormatter has a scale dictionary that maps the scale of the tick (the distance in days
    between one major tick) and a format string. The default looks like this:

```python
self.scaled = {
    365.0  : '%Y',
    30.   : '%b %Y',
    1.0   : '%b %d %Y',
    1./24. : '%H:%M:%S',
    1. / (24. * 60.) : '%H:%M:%S.%f',
}
```

The algorithm picks the key in the dictionary that is >= the current scale and uses that format string.
You can customize this dictionary by doing:

```python
>>> formatter = AutoDateFormatter()
>>> formatter.scaled[1/(24.*60.)] = '%M:%S' # only show min and sec
```

A custom `FuncFormatter` can also be used. The following example shows how to use a custom
format function to strip trailing zeros from decimal seconds and adds the date to the first ticklabel:

```python
>>> def my_format_function(x, pos=None):
...     x = matplotlib.dates.num2date(x)
...     if pos == 0:
...         fmt = '%D %H:%M:%S.%f'
...     else:
...         fmt = '%H:%M:%S.%f'
```
... label = x.strftime(fmt)
... label = label.rstrip("0")
... label = label.rstrip(".")
... return label
>>> from matplotlib.ticker import FuncFormatter
>>> formatter.scaled[1/(24.*60.)] = FuncFormatter(my_format_function)

Autoformat the date labels. The default format is the one to use if none of the values in self.scaled are greater than the unit returned by locator._get_unit().

```python
class matplotlib.dates.DateLocator(tz=None)
    Bases: matplotlib.ticker.Locator

tz is a tzinfo instance.

datalim_to_dt()

hms0d = {u'byminute': 0, u'byhour': 0, u'bysecond': 0}
	nonsingular(vmin, vmax)
    Given the proposed upper and lower extent, adjust the range if it is too close to being singular (i.e. a range of ~0).

set_tzinfo(tz)

viewlim_to_dt()
```

```python
class matplotlib.dates.RRuleLocator(o, tz=None)
    Bases: matplotlib.dates.DateLocator

autoscale()
    Set the view limits to include the data range.

static get_unit_generic(freq)
```

```python
class matplotlib.dates.AutoDateLocator(tz=None, minticks=5, maxticks=None, interval_multiples=False)
    Bases: matplotlib.dates.DateLocator

On autoscale, this class picks the best DateLocator to set the view limits and the tick locations.

minticks is the minimum number of ticks desired, which is used to select the type of ticking (yearly, monthly, etc.).

maxticks is the maximum number of ticks desired, which controls any interval between ticks (ticking every other, every 3, etc.). For really fine-grained control, this can be a dictionary mapping individual rrule frequency constants (YEARLY, MONTHLY, etc.) to their own maximum number of ticks. This can be used to keep the number of ticks appropriate to the format chosen in AutoDateFormatter. Any frequency not specified in this dictionary is given a default value.

tz is a tzinfo instance.
```
interval_multiples is a boolean that indicates whether ticks should be chosen to be multiple of the interval. This will lock ticks to ‘nicer’ locations. For example, this will force the ticks to be at hours 0, 6, 12, 18 when hourly ticking is done at 6 hour intervals.

The AutoDateLocator has an interval dictionary that maps the frequency of the tick (a constant from dateutil.rrule) and a multiple allowed for that ticking. The default looks like this:

```python
self.intervald = {
    YEARLY : [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500, 1000, 2000, 4000, 5000, 10000],
    MONTHLY : [1, 2, 3, 4, 6],
    DAILY : [1, 2, 3, 7, 14],
    HOURLY : [1, 2, 3, 4, 6, 12],
    MINUTELY: [1, 5, 10, 15, 30],
    SECONDLY: [1, 5, 10, 15, 30],
    MICROSECONDLY: [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, 1000000],
}
```

The interval is used to specify multiples that are appropriate for the frequency of ticking. For instance, every 7 days is sensible for daily ticks, but for minutes/seconds, 15 or 30 make sense. You can customize this dictionary by doing:

```python
locator = AutoDateLocator()
locator.intervald[HOURLY] = [3]  # only show every 3 hours
```

autoscale()

Try to choose the view limits intelligently.

get_locator(dmin, dmax)

Pick the best locator based on a distance.

nonsingular(vmin, vmax)

refresh()

Refresh internal information based on current limits.

set_axis(axis)

class matplotlib.dates.YearLocator(base=1, month=1, day=1, tz=None)

Bases: matplotlib.dates.DateLocator

Make ticks on a given day of each year that is a multiple of base.

Examples:

```python
# Tick every year on Jan 1st
locator = YearLocator()

# Tick every 5 years on July 4th
locator = YearLocator(5, month=7, day=4)
```
Mark years that are multiple of base on a given month and day (default jan 1).

```
autoscale()
```

Set the view limits to include the data range.

```
class matplotlib.dates.MonthLocator(bymonth=None, bymonthday=1, interval=1, tz=None)
```

Make ticks on occurances of each month month, e.g., 1, 3, 12.

Mark every month in `bymonth`; `bymonth` can be an int or sequence. Default is `range(1,13)`, i.e. every month.

`interval` is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

```
class matplotlib.dates.WeekdayLocator(byweekday=1, interval=1, tz=None)
```

Make ticks on occurances of each weekday.

Mark every weekday in `byweekday`; `byweekday` can be a number or sequence.

Elements of `byweekday` must be one of MO, TU, WE, TH, FR, SA, SU, the constants from `dateutil.rrule`, which have been imported into the `matplotlib.dates` namespace.

`interval` specifies the number of weeks to skip. For example, `interval=2` plots every second week.

```
class matplotlib.dates.DayLocator(bymonthday=None, interval=1, tz=None)
```

Make ticks on occurances of each day of the month. For example, 1, 15, 30.

Mark every day in `bymonthday`; `bymonthday` can be an int or sequence.

Default is to tick every day of the month: `bymonthday=range(1,32)`

```
class matplotlib.dates.HourLocator(byhour=None, interval=1, tz=None)
```

Make ticks on occurances of each hour.

Mark every hour in `byhour`; `byhour` can be an int or sequence. Default is to tick every hour: `byhour=range(24)`

`interval` is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

```
class matplotlib.dates.MinuteLocator(byminute=None, interval=1, tz=None)
```

Make ticks on occurances of each minute.

Mark every minute in `byminute`; `byminute` can be an int or sequence. Default is to tick every minute: `byminute=range(60)`
interval is the interval between each iteration. For example, if interval=2, mark every second occurrence.

class matplotlib.dates.SecondLocator(bysecond=None, interval=1, tz=None)
Bases: matplotlib.dates.RRuleLocator

Make ticks on occurrences of each second.

Mark every second in bysecond; bysecond can be an int or sequence. Default is to tick every second: bysecond = range(60)

interval is the interval between each iteration. For example, if interval=2, mark every second occurrence.

class matplotlib.dates.MicrosecondLocator(interval=1, tz=None)
Bases: matplotlib.dates.DateLocator

Make ticks on occurrences of each microsecond.

interval is the interval between each iteration. For example, if interval=2, mark every second microsecond.

set_axis(axis)

set_data_interval(vmin, vmax)

set_view_interval(vmin, vmax)

class matplotlib.dates.rrule(freq, dtstart=None, interval=1, wkst=None, count=None, until=None, bysetpos=None, bymonth=None, bymonthday=None, byyearday=None, byeaster=None, byweekday=None, byweekno=None, byweekday=None, byhour=None, byminute=None, bysecond=None, cache=False)
Bases: dateutil.rrule.rrulebase

That’s the base of the rule operation. It accepts all the keywords defined in the RFC as its constructor parameters (except byday, which was renamed to byweekday) and more. The constructor prototype is:

rrule(freq)

Where freq must be one of YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, or SECONDLY.

Additionally, it supports the following keyword arguments:

Parameters

- cache – If given, it must be a boolean value specifying to enable or disable caching of results. If you will use the same rrule instance multiple times, enabling caching will improve the performance considerably.
• **dtstart** – The recurrence start. Besides being the base for the recurrence, missing parameters in the final recurrence instances will also be extracted from this date. If not given, datetime.now() will be used instead.

• **interval** – The interval between each freq iteration. For example, when using YEARLY, an interval of 2 means once every two years, but with HOURLY, it means once every two hours. The default interval is 1.

• **wkst** – The week start day. Must be one of the MO, TU, WE constants, or an integer, specifying the first day of the week. This will affect recurrences based on weekly periods. The default week start is got from calendar.firstweekday(), and may be modified by calendar.setfirstweekday().

• **count** – How many occurrences will be generated.

• **until** – If given, this must be a datetime instance, that will specify the limit of the recurrence. If a recurrence instance happens to be the same as the datetime instance given in the until keyword, this will be the last occurrence.

• **bysetpos** – If given, it must be either an integer, or a sequence of integers, positive or negative. Each given integer will specify an occurrence number, corresponding to the nth occurrence of the rule inside the frequency period. For example, a bysetpos of -1 if combined with a MONTHLY frequency, and a byweekday of (MO, TU, WE, TH, FR), will result in the last work day of every month.

• **bymonth** – If given, it must be either an integer, or a sequence of integers, meaning the months to apply the recurrence to.

• **bymonthday** – If given, it must be either an integer, or a sequence of integers, meaning the month days to apply the recurrence to.

• **byyearday** – If given, it must be either an integer, or a sequence of integers, meaning the year days to apply the recurrence to. Week numbers have the meaning described in ISO8601, that is, the first week of the year is that containing at least four days of the new year.

• **byweekno** – If given, it must be either an integer, or a sequence of integers, meaning the week numbers to apply the recurrence to. Week numbers have the meaning described in ISO8601, that is, the first week of the year is that containing at least four days of the new year.

• **byweekday** – If given, it must be either an integer (0 == MO), a sequence of integers, one of the weekday constants (MO, TU, etc), or a sequence of these constants. When given, these variables will define the weekdays where the recurrence will be applied. It's also possible to use an argument n for the weekday instances, which will mean the nth occurrence of this weekday in the period. For example, with MONTHLY, or with YEARLY and BYMONTH, using FR(+1) in byweekday will specify the first friday of the month where the recurrence happens. Notice that in the RFC documentation, this is specified as BYDAY, but was renamed to avoid the ambiguity of that keyword.

• **byhour** – If given, it must be either an integer, or a sequence of integers, meaning the hours to apply the recurrence to.

• **bymminute** – If given, it must be either an integer, or a sequence of integers, meaning the minutes to apply the recurrence to.

• **bysecond** – If given, it must be either an integer, or a sequence of integers, meaning the seconds to apply the recurrence to.

• **byeaster** – If given, it must be either an integer, or a sequence of integers, positive or negative. Each integer will define an offset from the Easter Sun-
day. Passing the offset 0 to byeaster will yield the Easter Sunday itself. This is an extension to the RFC specification.

```python
class matplotlib.dates.relativeDelta(dt1=None, dt2=None, years=0, months=0, days=0, leapdays=0, weeks=0, hours=0, minutes=0, seconds=0, microseconds=0, year=None, month=None, day=None, weekday=None, yearday=None, nlyeaday=None, hour=None, minute=None, second=None, microsecond=None)
```

Bases: object

The relativedelta type is based on the specification of the excellent work done by M.-A. Lemburg in his mx.DateTime extension. However, notice that this type does NOT implement the same algorithm as his work. Do NOT expect it to behave like mx.DateTime’s counterpart.

There are two different ways to build a relativedelta instance. The first one is passing it two date/datetime classes:

```python
relativedelta(datetime1, datetime2)
```

The second one is passing it any number of the following keyword arguments:

```python
relativedelta(arg1=x,arg2=y,arg3=z...)
```

year, month, day, hour, minute, second, microsecond:
Absolute information (argument is singular); adding or subtracting a relativedelta with absolute information does not perform an arithmetic operation, but rather REPLACES the corresponding value in the original datetime with the value(s) in relativedelta.

years, months, weeks, days, hours, minutes, seconds, microseconds:
Relative information, may be negative (argument is plural); adding or subtracting a relativedelta with relative information performs the corresponding arithmetic operation on the original datetime value with the information in the relativedelta.

weekday:
One of the weekday instances (MO, TU, etc). These instances may receive a parameter N, specifying the Nth weekday, which could be positive or negative (like MO(+1) or MO(-2). Not specifying it is the same as specifying +1. You can also use an integer, where 0=MO.

leapdays:
Will add given days to the date found, if year is a leap year, and the date found is post 28 of february.

yearday, nlyeaday:
Set the yearday or the non-leap year day (jump leap days). These are converted to day/month/leapdays information.

Here is the behavior of operations with relativedelta:
1. Calculate the absolute year, using the ‘year’ argument, or the original datetime year, if the argument is not present.
2. Add the relative ‘years’ argument to the absolute year.
3. Do steps 1 and 2 for month/months.
4. Calculate the absolute day, using the ‘day’ argument, or the original datetime day, if the argument is not present. Then, subtract from the day until it fits in the year and month found after their operations.
5. Add the relative ‘days’ argument to the absolute day. Notice that the ‘weeks’ argument is multiplied by 7 and added to ‘days’.
6. Do steps 1 and 2 for hour/hours, minute/minutes, second/seconds, microsecond/microseconds.
7. If the ‘weekday’ argument is present, calculate the weekday, with the given (wday, nth) tuple. wday is the index of the weekday (0-6, 0=Mon), and nth is the number of weeks to add forward or backward, depending on its signal. Notice that if the calculated date is already Monday, for example, using (0, 1) or (0, -1) won’t change the day.

```python
matplotlib.dates.seconds(s)
    Return seconds as days.

matplotlib.dates.minutes(m)
    Return minutes as days.

matplotlib.dates.hours(h)
    Return hours as days.

matplotlib.dates.weeks(w)
    Return weeks as days.
```
48.1 matplotlib.dviread

An experimental module for reading dvi files output by TeX. Several limitations make this not (currently) useful as a general-purpose dvi preprocessor, but it is currently used by the pdf backend for processing usetex text.

Interface:

```python
dvi = Dvi(filename, 72)
# iterate over pages (but only one page is supported for now):
for page in dvi:
    w, h, d = page.width, page.height, page.descent
    for x, y, font, glyph, width in page.text:
        fontname = font.texname
        pointsize = font.size
        ...
    for x, y, height, width in page.boxes:
        ...
```

```python
class matplotlib.dviread.Dvi(filename, dpi)
Bases: object

A dvi (“device-independent”) file, as produced by TeX. The current implementation only reads the first page and does not even attempt to verify the postamble.

Initialize the object. This takes the filename as input and opens the file; actually reading the file happens when iterating through the pages of the file.

```close()
Close the underlying file if it is open.
```

```python
class matplotlib.dviread.DviFont(scale, tfm, texname, vf)
Bases: object

Object that holds a font’s texname and size, supports comparison, and knows the widths of glyphs in the same units as the AFM file. There are also internal attributes (for use by dviread.py) that are not used for comparison.

The size is in Adobe points (converted from TeX points).
```
texname
Name of the font as used internally by TeX and friends. This is usually very different from any external font names, and `dviread.PsfontsMap` can be used to find the external name of the font.

size
Size of the font in Adobe points, converted from the slightly smaller TeX points.

widths
Widths of glyphs in glyph-space units, typically 1/1000ths of the point size.

size
texname
widths

class matplotlib.dviread.Encoding(filename)
Bases: object
 Parses a *.enc file referenced from a psfonts.map style file. The format this class understands is a very limited subset of PostScript.

Usage (subject to change):

```python
for name in Encoding(filename):
    whatever(name)
```

encoding
class matplotlib.dviread.PsfontsMap(filename)
Bases: object
 A psfonts.map formatted file, mapping TeX fonts to PS fonts. Usage:

```python
>>> map = PsfontsMap(find_tex_file('pdftex.map'))
>>> entry = map['ptmbo8r']
>>> entry.texname
'ptmbo8r'
>>> entry.psname
'Times-Bold'
>>> entry.encoding
'/usr/local/texlive/2008/texmf-dist/fonts/enc/dvips/base/8r.enc'
>>> entry.effects
{'slant': 0.16700000000000001}
>>> entry.filename
```

For historical reasons, TeX knows many Type-1 fonts by different names than the outside world. (For one thing, the names have to fit in eight characters.) Also, TeX’s native fonts are not Type-1 but Metafont, which is nontrivial to convert to PostScript except as a bitmap. While high-quality
conversions to Type-1 format exist and are shipped with modern TeX distributions, we need to know which Type-1 fonts are the counterparts of which native fonts. For these reasons a mapping is needed from internal font names to font file names.

A texmf tree typically includes mapping files called e.g. psfonts.map, pdftex.map, dvipdfm.map. psfonts.map is used by dvips, pdftex.map by pdfTeX, and dvipdfm.map by dvipdfm. psfonts.map might avoid embedding the 35 PostScript fonts (i.e., have no filename for them, as in the Times-Bold example above), while the pdf-related files perhaps only avoid the “Base 14” pdf fonts. But the user may have configured these files differently.

class matplotlib.dviread.Tfm(filename)
   Bases: object

   A TeX Font Metric file. This implementation covers only the bare minimum needed by the Dvi class.

   checksum
      Used for verifying against the dvi file.

   design_size
      Design size of the font (in what units?)

   width
      Width of each character, needs to be scaled by the factor specified in the dvi file. This is a dict because indexing may not start from 0.

   height
      Height of each character.

   depth
      Depth of each character.

   checksum

   depth

   design_size

   height

   width

class matplotlib.dviread.Vf(filename)
   Bases: matplotlib.dviread.Dvi

   A virtual font (*.vf file) containing subroutines for dvi files.

   Usage:
vf = Vf(filename)
glyph = vf[code]
glyph.text, glyph.boxes, glyph.width

matplotlib.dviread.find_tex_file(filename, format=None)

Call kpsewhich to find a file in the texmf tree. If format is not None, it is used as the value for the --format option.

Apparently most existing TeX distributions on Unix-like systems use kpathsea. I hear MikTeX (a popular distribution on Windows) doesn’t use kpathsea, so what do we do? (TODO)

See also:

Kpathsea documentation The library that kpsewhich is part of.
49.1 matplotlib.figure

The figure module provides the top-level Artist, the Figure, which contains all the plot elements. The following classes are defined

SubplotParams control the default spacing of the subplots

Figure top level container for all plot elements

class matplotlib.figure.AxesStack
    Bases: matplotlib.cbook.Stack

Specialization of the Stack to handle all tracking of Axes in a Figure. This stack stores key, (ind, axes) pairs, where:

- key should be a hash of the args and kwargs used in generating the Axes.
- ind is a serial number for tracking the order in which axes were added.

The AxesStack is a callable, where ax_stack() returns the current axes. Alternatively the current_key_axes() will return the current key and associated axes.

add(key, a)
    Add Axes a, with key key, to the stack, and return the stack.

    If a is already on the stack, don’t add it again, but return None.

as_list()
    Return a list of the Axes instances that have been added to the figure

bubble(a)
    Move the given axes, which must already exist in the stack, to the top.

current_key_axes()
    Return a tuple of (key, axes) for the active axes.

    If no axes exists on the stack, then returns (None, None).

get(key)
    Return the Axes instance that was added with key. If it is not present, return None.

remove(a)
    Remove the axes from the stack.
class matplotlib.figure.Figure(figsize=None, dpi=None, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight_layout=None)

Bases: matplotlib.artist.Artist

The Figure instance supports callbacks through a callbacks attribute which is a matplotlib.cbook.CallbackRegistry instance. The events you can connect to are 'dpi_changed', and the callback will be called with func(fig) where fig is the Figure instance.

patch The figure patch is drawn by a matplotlib.patches.Rectangle instance

suppressComposite For multiple figure images, the figure will make composite images depending on the renderer option_image_nocomposite function. If suppressComposite is True|False, this will override the renderer.

figsize w,h tuple in inches

dpi Dots per inch

facecolor The figure patch facecolor; defaults to rc figure.facecolor

edgecolor The figure patch edge color; defaults to rc figure.edgecolor

linewidth The figure patch edge linewidth; the default linewidth of the frame

frameon If False, suppress drawing the figure frame

subplotpars A SubplotParams instance, defaults to rc

tight_layout If False use subplotpars; if True adjust subplot parameters using tight_layout() with default padding. When providing a dict containing the keys pad, w_pad, h_pad and rect, the default tight_layout() padings will be overridden. Defaults to rc figure.autolayout.

add_axes(*args, **kwargs)

Add an axes at position rect [left, bottom, width, height] where all quantities are in fractions of figure width and height. kwargs are legal Axes kwargs plus projection which sets the projection type of the axes. (For backward compatibility, polar=True may also be provided, which is equivalent to projection='polar'). Valid values for projection are: ['aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear']. Some of these projections support additional kwargs, which may be provided to add_axes(). Typical usage:

rect = l,b,w,h
fig.add_axes(rect)
fig.add_axes(rect, frameon=False, axisbg='g')
fig.add_axes(rect, polar=True)
fig.add_axes(rect, projection='polar')
fig.add_axes(ax)

If the figure already has an axes with the same parameters, then it will simply make that axes current and return it. If you do not want this behavior, e.g., you want to force the creation of a new Axes, you must use a unique set of args and kwargs. The axes label attribute has been exposed for this purpose. e.g., if you want two axes that are otherwise identical to be added to the figure, make sure you give them unique labels:

fig.add_axes(rect, label='axes1')
fig.add_axes(rect, label='axes2')

In rare circumstances, add_axes may be called with a single argument, an Axes instance already created in the present figure but not in the figure’s list of axes. For example, if an axes has been removed with delaxes(), it can be restored with:
fig.add_axes(ax)

In all cases, the `Axes` instance will be returned.

In addition to `projection`, the following kwargs are supported:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adjustable</code></td>
<td>[‘box’</td>
</tr>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>anchor</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>aspect</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>autoscale_on</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>autoscalex_on</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>autoscaley_on</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>axes_locator</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axis_bgcolor</code></td>
<td>any matplotlib color - see <code>colors()</code></td>
</tr>
<tr>
<td><code>axisbelow</code></td>
<td>[ True</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color_cycle</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>frame_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>navigate</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>navigate_mode</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterization_zorder</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>title</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>xbound</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>xlabel</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>xlim</code></td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td><code>xmargin</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
add_axobserver(func)

whenever the axes state change, func(self) will be called

add_subplot(*args, **kwargs)

Add a subplot. Examples:

    fig.add_subplot(111)

    # equivalent but more general
    fig.add_subplot(1,1,1)

    # add subplot with red background
    fig.add_subplot(212, axisbg='r')

    # add a polar subplot
    fig.add_subplot([111, projection='polar'])

    # add Subplot instance sub
    fig.add_subplot(sub)

kwargs are legal Axes kwargs plus projection, which chooses a projection type for the axes.
(For backward compatibility, polar=True may also be provided, which is equivalent to projection='polar'). Valid values for projection are: [u’aitoff’, u’hammer’, u’lambert’, u’mollweide’, u’polar’, u’rectilinear’]. Some of these projections support additional kwargs, which may be provided to add_axes().

The Axes instance will be returned.

If the figure already has a subplot with key (args, kwargs) then it will simply make that subplot current and return it.

See also:

subplot() for an explanation of the args.

The following kwargs are supported:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>[‘box’</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axis_bgcolor</td>
<td>any matplotlib color - see colors()</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>navigate</td>
<td>[True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
<tr>
<td>xscale</td>
<td>[u‘linear’</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 49.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ylim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>[u’linear’</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**autofmt_xdate** *(bottom=0.2, rotation=30, ha=u’right’)*

Date ticklabels often overlap, so it is useful to rotate them and right align them. Also, a common use case is a number of subplots with shared xaxes where the x-axis is date data. The ticklabels are often long, and it helps to rotate them on the bottom subplot and turn them off on other subplots, as well as turn off xlabels.

*bottom* The bottom of the subplots for `subplots_adjust()`

*rotation* The rotation of the xtick labels

*ha* The horizontal alignment of the xticklabels

**axes**

Read-only: list of axes in Figure

**clear()**

Clear the figure – synonym for `clf()`.

**clf(keep_observers=False)**

Clear the figure.

Set `keep_observers` to True if, for example, a gui widget is tracking the axes in the figure.

**colorbar(mappable, cax=None, ax=None, use_gridspec=True, **kw)**

Create a colorbar for a ScalarMappable instance, `mappable`.

Documentation for the pylab thin wrapper:

Add a colorbar to a plot.

Function signatures for the `pyplot` interface; all but the first are also method signatures for the `colorbar()` method:

```
colorbar(**kwargs)
colorbar(mappable, **kwargs)
colorbar(mappable, cax=cax, **kwargs)
colorbar(mappable, ax=ax, **kwargs)
```

arguments:

*mappable* the `Image`, `ContourSet`, etc. to which the colorbar applies; this argument is mandatory for the `colorbar()` method but optional for the `colorbar()` function, which sets the default to the current image.

keyword arguments:

* cax None | axes object into which the colorbar will be drawn
ax None | parent axes object(s) from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.

use_gridspec False | If cax is None, a new cax is created as an instance of Axes. If ax is an instance of Subplot and use_gridspec is True, cax is created as an instance of Subplot using the grid_spec module.

Additional keyword arguments are of two kinds:

**axes properties:**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>panchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

**colorbar properties:**
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extend</td>
<td>[ ‘neither’</td>
</tr>
<tr>
<td>extendfrac</td>
<td>[ None</td>
</tr>
<tr>
<td>extendrect</td>
<td>[ False</td>
</tr>
<tr>
<td>spacing</td>
<td>[ ‘uniform’</td>
</tr>
<tr>
<td>ticks</td>
<td>[ None</td>
</tr>
<tr>
<td>format</td>
<td>[ None</td>
</tr>
<tr>
<td>drawedges</td>
<td>[ False</td>
</tr>
</tbody>
</table>

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundaries</td>
<td>None or a sequence</td>
</tr>
<tr>
<td>values</td>
<td>None or a sequence which must be of length 1 less than the sequence of boundaries. For each region delimited by adjacent entries in boundaries, the color mapped to the corresponding value in values will be used.</td>
</tr>
</tbody>
</table>

If mappable is a ContourSet, its extend kwarg is included automatically.

Note that the shrink kwarg provides a simple way to keep a vertical colorbar, for example, from being taller than the axes of the mappable to which the colorbar is attached; but it is a manual
method requiring some trial and error. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of `shrink`.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties `kwargs`.

It is known that some vector graphics viewer (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers not matplotlib. As a workaround the colorbar can be rendered with overlapping segments:

```python
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances. Particularly with semi transparent images (alpha < 1) and colorbar extensions and is not enabled by default see (issue #1188).

`returns: Colorbar` instance; see also its base class, `ColorbarBase`. Call the `set_label()` method to label the colorbar.

`contains(mouseevent)`
Test whether the mouse event occurred on the figure.

Returns True,{}

`delaxes(a)`
remove a from the figure and update the current axes

`dpi`

`draw(artist, renderer, *args, **kwargs)`
Render the figure using `matplotlib.backend_bases.RendererBase` instance `renderer`.

`draw_artist(a)`

`draw` `matplotlib.artist.Artist` instance `a` only – this is available only after the figure is drawn

`figimage(X, xo=0, yo=0, alpha=None, norm=None, cmap=None, vmin=None, vmax=None, origin=None, **kwargs)`
Adds a non-resampled image to the figure.

call signatures:

`figimage(X, **kwargs)`

adds a non-resampled array `X` to the figure.

`figimage(X, xo, yo)`

with pixel offsets `xo, yo`,

`X` must be a float array:
If $X$ is $M \times N$, assume luminance (grayscale)

- If $X$ is $M \times N \times 3$, assume RGB
- If $X$ is $M \times N \times 4$, assume RGBA

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xo or yo</td>
<td>An integer, the $x$ and $y$ image offset in pixels</td>
</tr>
<tr>
<td>cmap</td>
<td>a <code>matplotlib.colors.Colormap</code> instance, e.g., <code>cm.jet</code>. If <code>None</code>, default to the rc <code>image.cmap</code> value</td>
</tr>
<tr>
<td>norm</td>
<td>a <code>matplotlib.colors.Normalize</code> instance. The default is <code>normalization()</code> (default) which scales luminance -&gt; 0-1</td>
</tr>
<tr>
<td>vmin, vmax</td>
<td>Used to scale a luminance image to 0-1. If either is <code>None</code>, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for $vmin$ and $vmax$ will be ignored.</td>
</tr>
<tr>
<td>alpha</td>
<td>the alpha blending value, default is <code>None</code></td>
</tr>
<tr>
<td>origin</td>
<td>`[‘upper’</td>
</tr>
</tbody>
</table>

`figimage` complements the axes image (`imshow()`) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an `Axes` with size `[0,1,0,1]`.

An `matplotlib.image.FigureImage` instance is returned.
Additional kwargs are Artist kwargs passed on to FigureImage

**gca(**kwargs)**

Get the current axes, creating one if necessary

The following kwargs are supported for ensuring the returned axes adheres to the given projection etc., and for axes creation if the active axes does not exist:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>['box'</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbgcolor</td>
<td>any matplotlib color - see colors()</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[ True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 49.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_path</td>
<td>[Path, Transform]</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True, False]</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True, False]</td>
</tr>
<tr>
<td>navigate</td>
<td>[True, False]</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
<tr>
<td>xscale</td>
<td>[u'linear', u'log', u'symlog']</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>[u'linear', u'log', u'symlog']</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_axes()

get_children()

get_default_bbox_extra_artists()
get_dpi()  
Return the dpi as a float

get_edgecolor()  
Get the edge color of the Figure rectangle

get_facecolor()  
Get the face color of the Figure rectangle

get_figheight()  
Return the figheight as a float

get_figwidth()  
Return the figwidth as a float

get_frameon()  
get the boolean indicating frameon

get_size_inches()  
Returns the current size of the figure in inches (1in == 2.54cm) as an numpy array.  
Returns size : ndarray  
The size of the figure in inches  
See also:  
matplotlib.Figure.set_size_inches

get_tight_layout()  
Return the Boolean flag, True to use :meth:`tight_layout` when drawing.

get_tightbbox(renderer)  
Return a (tight) bounding box of the figure in inches.  
It only accounts axes title, axis labels, and axis ticklabels. Needs improvement.

get_window_extent(*args, **kwargs)  
get the figure bounding box in display space; kwargs are void

ginput(n=1, timeout=30, show_clicks=True, mouse_add=1, mouse_pop=3, mouse_stop=2)  
Call signature:

```python  
ginput(self, n=1, timeout=30, show_clicks=True, mouse_add=1, mouse_pop=3, mouse_stop=2)  
```

Blocking call to interact with the figure.
This will wait for $n$ clicks from the user and return a list of the coordinates of each click.
If $timeout$ is zero or negative, does not timeout.
If $n$ is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons at once) terminates the input.
Right clicking cancels last input.
The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overridden via the arguments `mouse_add`, `mouse_pop` and `mouse_stop`, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

**hold**(*b=None*)

Set the hold state. If hold is None (default), toggle the hold state. Else set the hold state to boolean value b.

e.g.:

```python
hold()  # toggle hold
hold(True)  # hold is on
hold(False)  # hold is off
```

**legend**(*handles, labels, *args, **kwargs*)

Place a legend in the figure. Labels are a sequence of strings, handles is a sequence of Line2D or Patch instances, and loc can be a string or an integer specifying the legend location.

**USAGE:**

```python
legend( (line1, line2, line3),
       ('label1', 'label2', 'label3'),
       'upper right')
```

The `loc` location codes are:

- `'best'`: 0,  
- `'upper right'`: 1,  
- `'upper left'`: 2,  
- `'lower left'`: 3,  
- `'lower right'`: 4,  
- `'right'`: 5,  
- `'center left'`: 6,  
- `'center right'`: 7,  
- `'lower center'`: 8,  
- `'upper center'`: 9,  
- `'center'`: 10,

`loc` can also be an (x,y) tuple in figure coords, which specifies the lower left of the legend box.  
figure coords are (0,0) is the left, bottom of the figure and 1,1 is the right, top.

Keyword arguments:

- `prop`: [ *None* | FontProperties | dict ] A `matplotlib.font_manager.FontProperties` instance. If `prop` is a dictionary, a new instance will be created with `prop`. If *None*, use rc settings.

- `numpoints`: integer The number of points in the legend line, default is 4

- `scatterpoints`: integer The number of points in the legend line, default is 4
scatteryoffsets: list of floats a list of yoffsets for scatter symbols in legend

markerscale: [ None | scalar ] The relative size of legend markers vs. original. If None, use rc settings.

fancybox: [ None | False | True ] if True, draw a frame with a round fancybox. If None, use rc

shadow: [ None | False | True ] If True, draw a shadow behind legend. If None, use rc settings.

ncol [integer] number of columns. default is 1

mode [[ “expand” | None ]] if mode is “expand”, the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor)

title [string] the legend title

Padding and spacing between various elements use following keywords parameters. The dimensions of these values are given as a fraction of the fontsize. Values from rcParams will be used if None.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>borderpad</td>
<td>the fractional whitespace inside the legend border</td>
</tr>
<tr>
<td>labelspacing</td>
<td>the vertical space between the legend entries</td>
</tr>
<tr>
<td>handlelength</td>
<td>the length of the legend handles</td>
</tr>
<tr>
<td>handletextpad</td>
<td>the pad between the legend handle and text</td>
</tr>
<tr>
<td>borderaxespad</td>
<td>the pad between the axes and legend border</td>
</tr>
<tr>
<td>columnspacing</td>
<td>the spacing between columns</td>
</tr>
</tbody>
</table>

Note: Not all kinds of artist are supported by the legend. See LINK (FIXME) for details.

Example:
**savefig(***, *args, **kwargs*)

Save the current figure.

Call signature:

```python
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
orientation='portrait', papertype=None, format=None,
transparent=False, bbox_inches=None, pad_inches=0.1,
frameon=None)
```

The output formats available depend on the backend being used.

Arguments:

- **fname**: A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as `PdfPages`.

  If `format` is `None` and `fname` is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter `savefig.format` is used.

  If `fname` is not a string, remember to specify `format` to ensure that the correct backend is used.

Keyword arguments:

- **dpi**: `[None | scalar > 0]` The resolution in dots per inch. If `None` it will default to the value `savefig.dpi` in the matplotlibrc file.
**facecolor, edgcolor:** the colors of the figure rectangle

**orientation:** ['landscape', 'portrait'] not supported on all backends; currently only on postscript output


**format:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

**transparent:** If True, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgcolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

**frameon:** If True, the figure patch will be colored, if False, the figure background will be transparent. If not provided, the rcParam ‘savefig.frameon’ will be used.

**bbox_inches:** Bbox in inches. Only the given portion of the figure is saved. If ‘tight’, try to figure out the tight bbox of the figure.

**pad_inches:** Amount of padding around the figure when bbox_inches is ‘tight’.

**bbox_extra_artists:** A list of extra artists that will be considered when the tight bbox is calculated.

**sca(a)**
Set the current axes to be a and return a

**set_canvas(canvas)**
Set the canvas the contains the figure

ACCEPTS: a FigureCanvas instance

**set_dpi(val)**
Set the dots-per-inch of the figure

ACCEPTS: float

**set_edgecolor(color)**
Set the edge color of the Figure rectangle

ACCEPTS: any matplotlib color - see help(colors)

**set_facecolor(color)**
Set the face color of the Figure rectangle

ACCEPTS: any matplotlib color - see help(colors)

**set_figheight(val)**
Set the height of the figure in inches

ACCEPTS: float

**set_figwidth(val)**
Set the width of the figure in inches

ACCEPTS: float
**set_frameon**(\(b\))
Set whether the figure frame (background) is displayed or invisible

ACCEPTS: boolean

**set_size_inches**(\(w, h, forward=False\))
Set the figure size in inches (1in == 2.54cm)

Usage:

```python
fig.set_size_inches(w,h)  # OR
fig.set_size_inches((w,h))
```

optional kwarg \(forward=True\) will cause the canvas size to be automatically updated; e.g., you can resize the figure window from the shell

ACCEPTS: a \(w,h\) tuple with \(w,h\) in inches

**See also:**

matplotlib.Figure.get_size_inches

**set_tight_layout**(tight)
Set whether \(tight_layout()\) is used upon drawing. If None, the rc-Params[‘figure.autolayout’] value will be set.

When providing a dict containing the keys \(pad, w\_pad, h\_pad\) and \(rect\), the default \(tight_layout()\) paddings will be overridden.

ACCEPTS: [True | False | dict | None ]

**show**(warn=True)
If using a GUI backend with pyplot, display the figure window.

If the figure was not created using \(figure()\), it will lack a FigureManagerBase, and will raise an AttributeError.

For non-GUI backends, this does nothing, in which case a warning will be issued if \(warn\) is True (default).

**subplots_adjust**(\(*args, **kwargs\))
Call signature:

```python
subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
```

Update the SubplotParams with \(kwargs\) (defaulting to rc when \(None\)) and update the subplot locations

**suptitle**(\(t, **kwargs\))
Add a centered title to the figure.

\(kwargs\) are matplotlib.text.Text properties. Using figure coordinates, the defaults are:

- \(x\) [0.5] The x location of the text in figure coords
- \(y\) [0.98] The y location of the text in figure coords
**horizontalalignment** ['center'] The horizontal alignment of the text

**verticalalignment** ['top'] The vertical alignment of the text

A `matplotlib.text.Text` instance is returned.

Example:

```python
fig.suptitle('this is the figure title', fontsize=12)
```

**text**(x, y, s, *args, **kwargs)

Add text to figure.

Call signature:

```python
text(x, y, s, fontdict=None, **kwargs)
```

Add text to figure at location `x, y` (relative 0-1 coords). See `text()` for the meaning of the other arguments.

kwars control the `Text` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>['center'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left'</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>(x,y)</td>
</tr>
<tr>
<td>position</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
</tbody>
</table>
**Table 49.4 – continued from previous page**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>snap</code> or <code>fontstretch</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>stretch</code> or <code>fontstretch</code></td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td><code>style</code> or <code>fontstyle</code></td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><code>text</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>variant</code> or <code>fontvariant</code></td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><code>verticalalignment</code> or <code>va</code> or <code>ma</code></td>
<td>[ ‘center’</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>x</code></td>
<td>float</td>
</tr>
<tr>
<td><code>y</code></td>
<td>float</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**tight_layout** *(renderer=None, pad=1.08, h_pad=None, w_pad=None, rect=None)*

Adjust subplot parameters to give specified padding.

Parameters:

- `pad` [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

- `h_pad, w_pad` [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.

- `rect` [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

**waitforbuttonpress** *(timeout=-1)*

Call signature:

```
waitforbuttonpress(self, timeout=-1)
```

Blocking call to interact with the figure.

This will return True is a key was pressed, False if a mouse button was pressed and None if `timeout` was reached without either being pressed.

If `timeout` is negative, does not timeout.

**class matplotlib.figure.SubplotParams** *(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)*

A class to hold the parameters for a subplot

All dimensions are fraction of the figure width or height. All values default to their rc params

The following attributes are available

- `left` [0.125] The left side of the subplots of the figure

- `right` [0.9] The right side of the subplots of the figure
**bottom** [0.1] The bottom of the subplots of the figure

**top** [0.9] The top of the subplots of the figure

**wspace** [0.2] The amount of width reserved for blank space between subplots

**hspace** [0.2] The amount of height reserved for white space between subplots

**update** *(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)*

Update the current values. If any kwarg is None, default to the current value, if set, otherwise to rc

matplotlib.figure.figaspect(*arg*)

Create a figure with specified aspect ratio. If *arg* is a number, use that aspect ratio. If *arg* is an array, figaspect will determine the width and height for a figure that would fit array preserving aspect ratio. The figure width, height in inches are returned. Be sure to create an axes with equal with and height, e.g.,

Example usage:

```python
# make a figure twice as tall as it is wide
w, h = figaspect(2.)
fig = Figure(figsize=(w,h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)

# make a figure with the proper aspect for an array
A = rand(5,3)
w, h = figaspect(A)
fig = Figure(figsize=(w,h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Thanks to Fernando Perez for this function
50.1 matplotlib.finance

A collection of functions for collecting, analyzing and plotting financial data. User contributions welcome!
This module is deprecated in 1.4 and will be moved to mpl_toolkits or it's own project in the future.

```
matplotlib.finance.candlestick(ax, quotes, width=0.2, colorup=u'k', colordown=u'r', alpha=1.0)
```

Plot the time, open, close, high, low as a vertical line ranging from low to high. Use a rectangular bar to represent the open-close span. If close >= open, use colorup to color the bar, otherwise use colordown

This function has been deprecated in 1.4 in favor of `candlestick_ochl`, which maintains the original argument order, or `candlestick_ohlc`, which uses the open-high-low-close order. This function will be removed in 1.5

**Parameters**

- `ax` : Axes
  an Axes instance to plot to

- `quotes` : sequence of (time, open, close, high, low, ...) sequences
  As long as the first 5 elements are these values, the record can be as long as you want (e.g., it may store volume).

  time must be in float days format - see `date2num`

- `width` : float
  fraction of a day for the rectangle width

- `colorup` : color
  the color of the rectangle where close >= open

- `colordown` : color
  the color of the rectangle where close < open

- `alpha` : float
  the rectangle alpha level

**Returns**

- `ret` : tuple
  returns (lines, patches) where lines is a list of lines added and patches is a list of the rectangle patches added

```
matplotlib.finance.candlestick2(ax, opens, closes, highs, lows, width=4, colorup=u'k', colordown=u'r', alpha=0.75)
```

Represent the open, close as a bar line and high low range as a vertical line.
This function has been deprecated in 1.4 in favor of `candlestick2_ochl`, which maintains the original argument order, or `candlestick2_ohlc`, which uses the open-high-low-close order. This function will be removed in 1.5

**Parameters**

- `ax` : Axes
  - an Axes instance to plot to
- `opens` : sequence
  - sequence of opening values
- `closes` : sequence
  - sequence of closing values
- `highs` : sequence
  - sequence of high values
- `lows` : sequence
  - sequence of low values
- `ticksize` : int
  - size of open and close ticks in points
- `colorup` : color
  - the color of the lines where close >= open
- `colordown` : color
  - the color of the lines where close < open
- `alpha` : float
  - bar transparency

**Returns**

- `ret` : tuple
  - `(lineCollection, barCollection)`

`matplotlib.finance.candlestick2_ochl(ax, opens, closes, highs, lows, width=4, colorup=u'k', colordown=u'r', alpha=0.75)`

Represent the open, close as a bar line and high low range as a vertical line.

Preserves the original argument order.

**Parameters**

- `ax` : Axes
  - an Axes instance to plot to
- `opens` : sequence
  - sequence of opening values
- `closes` : sequence
  - sequence of closing values
- `highs` : sequence
  - sequence of high values
- `lows` : sequence
  - sequence of low values
- `ticksize` : int
  - size of open and close ticks in points
- `colorup` : color
  - the color of the lines where close >= open
- `colordown` : color
  - the color of the lines where close < open
- `alpha` : float
  - bar transparency

**Returns**

- `ret` : tuple
  - `(lineCollection, barCollection)`
```
from matplotlib.finance import candlestick2_ohlc

candlestick2_ohlc(ax, opens, highs, lows, closes, width=4, colorup=u'k', colordown=u'r', alpha=0.75)

Represent the open, close as a bar line and high low range as a vertical line.

**Parameters**

- `ax`: Axes
  - an Axes instance to plot to
- `opens`: sequence
  - sequence of opening values
- `highs`: sequence
  - sequence of high values
- `lows`: sequence
  - sequence of low values
- `closes`: sequence
  - sequence of closing values
- `ticksize`: int
  - size of open and close ticks in points
- `colorup`: color
  - the color of the lines where close >= open
- `colordown`: color
  - the color of the lines where close < open
- `alpha`: float
  - bar transparency

**Returns**

- `ret`: tuple
  - (lineCollection, barCollection)
```

```
from matplotlib.finance import candlestick_ochl

candlestick_ochl(ax, quotes, width=0.2, colorup=u'k', colordown=u'r', alpha=1.0)

Plot the time, open, close, high, low as a vertical line ranging from low to high. Use a rectangular bar to represent the open-close span. If close >= open, use colorup to color the bar, otherwise use colordown.

**Parameters**

- `ax`: Axes
  - an Axes instance to plot to
- `quotes`: sequence of (time, open, close, high, low, ...) sequences
  - As long as the first 5 elements are these values, the record can be as long as you want (e.g., it may store volume).
  - time must be in float days format - see date2num
- `width`: float
  - fraction of a day for the rectangle width
- `colorup`: color
  - the color of the rectangle where close >= open
- `colordown`: color
  - the color of the rectangle where close < open
- `alpha`: float
  - the rectangle alpha level

**Returns**

- `ret`: tuple
  - returns (lines, patches) where lines is a list of lines added and patches is a list of the rectangle patches added
```

```
from matplotlib.finance import candlestick_ohlc

candlestick_ohlc(ax, quotes, width=0.2, colorup=u'k', colordown=u'r', alpha=1.0)
```

Plot the time, open, high, low, close as a vertical line ranging from low to high. Use a rectangular bar to represent the open-close span. If close >= open, use colorup to color the bar, otherwise use colordown.

**Parameters**
- **ax**: Axes
  - an Axes instance to plot to
- **quotes**: sequence of (time, open, high, low, close, ...) sequences
  - As long as the first 5 elements are these values, the record can be as long as you want (e.g., it may store volume).
  - time must be in float days format - see date2num
- **width**: float
  - fraction of a day for the rectangle width
- **colorup**: color
  - the color of the rectangle where close >= open
- **colordown**: color
  - the color of the rectangle where close < open
- **alpha**: float
  - the rectangle alpha level

**Returns**
- **ret**: tuple
  - returns (lines, patches) where lines is a list of lines added and patches is a list of the rectangle patches added

```python
matplotlib.finance.fetch_historical_yahoo(ticker, date1, date2, cachename=None, dividends=False)
```

Fetch historical data for ticker between date1 and date2. date1 and date2 are date or datetime instances, or (year, month, day) sequences.

**Parameters**
- **ticker**: str
- **date1**: sequence of form (year, month, day), datetime, or date
- **date2**: sequence of form (year, month, day), datetime, or date
- **cachename**: str
  - cachename is the name of the local file cache. If None, will default to the md5 hash or the url (which incorporates the ticker and date range)
- **dividends**: bool
  - set dividends=True to return dividends instead of price data. With this option set, parse functions will not work

**Returns**
- **file_handle**: file handle
  - a file handle is returned

**Examples**

```python
>>> fh = fetch_historical_yahoo('^GSPC', (2000, 1, 1), (2001, 12, 31))
```

```python
matplotlib.finance.index_bar(ax, vals, facecolor='u'b', edgecolor='u'l', width=4, alpha=1.0)
```

Add a bar collection graph with height vals (-1 is missing).

**Parameters**
- **ax**: Axes
an Axes instance to plot to

vals : sequence
    a sequence of values

facecolor : color
    the color of the bar face

decolor : color
    the color of the bar edges

width : int
    the bar width in points

alpha : float
    bar transparency

Returns ret : barCollection
    The barCollection added to the axes

**matplotlib.finance.parse_yahoo_historical** *(fh, adjusted=True, asobject=False)*

Parse the historical data in file handle fh from Yahoo finance.

This function has been deprecated in 1.4 in favor of **parse_yahoo_historical_ochl**, which maintains the original argument order, or **parse_yahoo_historical_ohlc**, which uses the open-high-low-close order. This function will be removed in 1.5

**Parameters adjusted** : bool

If True (default) replace open, close, high, low prices with their adjusted values. The adjustment is by a scale factor, $S = \frac{\text{adjusted_close}}{\text{close}}$. Adjusted prices are actual prices multiplied by $S$.

Volume is not adjusted as it is already backward split adjusted by Yahoo. If you want to compute dollars traded, multiply volume by the adjusted close, regardless of whether you choose adjusted = True|False.

**asobject** : bool or None

If False (default for compatibility with earlier versions) return a list of tuples containing
d, open, close, high, low, volume

If None (preferred alternative to False), return a 2-D ndarray corresponding to the list of tuples.

Otherwise return a numpy recarray with
date, year, month, day, d, open, close, high, low, volume,
adjusted_close

where d is a floating poing representation of date, as returned by date2num, and date is a python standard library datetime.date instance.

The name of this kwarg is a historical artifact. Formerly, True returned a cbook Bunch holding 1-D ndarrays. The behavior of a numpy recarray is very similar to the Bunch.

**ochl** : bool

Temporary argument to select between ochl and ohlc ordering. De-faults to True to preserve original functionality.

**matplotlib.finance.parse_yahoo_historical_ochl** *(fh, adjusted=True, asobject=False)*

Parse the historical data in file handle fh from Yahoo finance.

**Parameters adjusted** : bool
If True (default) replace open, close, high, low prices with their adjusted values. The adjustment is by a scale factor, $S = \text{adjusted\_close}/\text{close}$. Adjusted prices are actual prices multiplied by $S$.

Volume is not adjusted as it is already backward split adjusted by Yahoo. If you want to compute dollars traded, multiply volume by the adjusted close, regardless of whether you choose adjusted = True|False.

**asobject** : bool or None
If False (default for compatibility with earlier versions) return a list of tuples containing
- $d$, open, close, high, low, volume
If None (preferred alternative to False), return a 2-D ndarray corresponding to the list of tuples.

Otherwise return a numpy recarray with
- date, year, month, day, $d$, open, close, high, low, volume, adjusted\_close
where $d$ is a floating point representation of date, as returned by `date2num`, and date is a python standard library `datetime.date` instance.

The name of this kwarg is a historical artifact. Formerly, True returned a cbook Bunch holding 1-D ndarrays. The behavior of a numpy recarray is very similar to the Bunch.

```
matplotlib.finance.parse_yahoo_historical_ohlc(fh, adjusted=True, asobject=False)
```

Parse the historical data in file handle fh from yahoo finance.

**Parameters adjusted** : bool
If True (default) replace open, high, low, close prices with their adjusted values. The adjustment is by a scale factor, $S = \text{adjusted\_close}/\text{close}$. Adjusted prices are actual prices multiplied by $S$.

Volume is not adjusted as it is already backward split adjusted by Yahoo. If you want to compute dollars traded, multiply volume by the adjusted close, regardless of whether you choose adjusted = True|False.

**asobject** : bool or None
If False (default for compatibility with earlier versions) return a list of tuples containing
- $d$, open, high, low, close, volume
If None (preferred alternative to False), return a 2-D ndarray corresponding to the list of tuples.

Otherwise return a numpy recarray with
- date, year, month, day, $d$, open, high, low, close, volume, adjusted\_close
where $d$ is a floating point representation of date, as returned by `date2num`, and date is a python standard library `datetime.date` instance.

The name of this kwarg is a historical artifact. Formerly, True returned a cbook Bunch holding 1-D ndarrays. The behavior of a numpy recarray is very similar to the Bunch.
**matplotlib.finance.plot_day_summary**

Plots day summary

Represent the time, open, close, high, low as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

This function has been deprecated in 1.4 in favor of **plot_day_summary_ochl**, which maintains the original argument order, or **plot_day_summary_ohlc**, which uses the open-high-low-close order. This function will be removed in 1.5

**Parameters**

- **ax**: Axes instance to plot to
- **quotes**: sequence of (time, open, close, high, low, ...) sequences data to plot. time must be in float date format - see date2num
- **ticksize**: int
  - open/close tick marker in points
- **colorup**: color
  - the color of the lines where close >= open
- **colordown**: color
  - the color of the lines where close < open

**Returns**

- **lines**: list
  - list of tuples of the lines added (one tuple per quote)

**matplotlib.finance.plot_day_summary2**

Represent the time, open, close, high, low, as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

This function has been deprecated in 1.4 in favor of **plot_day_summary2_ochl**, which maintains the original argument order, or **plot_day_summary2_ohlc**, which uses the open-high-low-close order. This function will be removed in 1.5

**Parameters**

- **ax**: Axes instance to plot to
- **opens**: sequence of opening values
- **closes**: sequence of closing values
- **highs**: sequence of high values
- **lows**: sequence of low values
- **ticksize**: int
  - size of open and close ticks in points
- **colorup**: color
  - the color of the lines where close >= open
- **colordown**: color
  - the color of the lines where close < open

**Returns**

- **ret**: list
  - a list of lines added to the axes

**matplotlib.finance.plot_day_summary2_ochl**


Represent the time, open, close, high, low, as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

**Parameters**

- **ax**: `Axes`  
  an Axes instance to plot to
- **opens**: sequence  
  sequence of opening values
- **closes**: sequence  
  sequence of closing values
- **highs**: sequence  
  sequence of high values
- **lows**: sequence  
  sequence of low values
- **ticksize**: int  
  size of open and close ticks in points
- **colorup**: color  
  the color of the lines where close >= open
- **colordown**: color  
  the color of the lines where close < open

**Returns**

- **ret**: list  
  a list of lines added to the axes

**Example**

```python
c mpl.finance.plot_day_summary2_ohlc(ax, opens, highs, lows, closes, ticksize=4, colorup=u’k’, colordown=u’r’)
```

Plots day summary

Represent the time, open, close, high, low, as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

**Parameters**

- **ax**: `Axes`  
  an Axes instance to plot to
- **quotes**: sequence  
  sequence of quotes
- **ticksize**: int  
  size of open and close ticks in points
- **colorup**: color  
  the color of the lines where close >= open
- **colordown**: color  
  the color of the lines where close < open

**Returns**

- **ret**: list  
  a list of lines added to the axes

**Example**

```python
c mpl.finance.plot_day_summary_ohlc(ax, opens, highs, lows, closes, ticksize=3, colorup=u’k’, colordown=u’r’)
```

Plots day summary

Represent the time, open, close, high, low, as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.
Parameters **ax** : *Axes*
   
an *Axes* instance to plot to

**quotes** : sequence of (time, open, close, high, low, ...) sequences
data to plot. time must be in float date format - see date2num

**ticksize** : int
   
open/close tick marker in points

**colorup** : color
   
the color of the lines where close $\geq$ open

**colordown** : color
   
the color of the lines where close $< open$

Returns **lines** : list
   
list of tuples of the lines added (one tuple per quote)

**matplotlib.finance.plot_day_summary_ohlc**(*ax*, **quotes**, **ticksize**=3, **colorup**=u'k', **colordown**=u'r')

Plots day summary
   
Represent the time, open, high, low, close as a vertical line ranging from low to high. The
left tick is the open and the right tick is the close.

Parameters **ax** : *Axes*
   
an *Axes* instance to plot to

**quotes** : sequence of (time, open, high, low, close, ...) sequences
data to plot. time must be in float date format - see date2num

**ticksize** : int
   
open/close tick marker in points

**colorup** : color
   
the color of the lines where close $\geq$ open

**colordown** : color
   
the color of the lines where close $< open$

Returns **lines** : list
   
list of tuples of the lines added (one tuple per quote)

**matplotlib.finance.quotes_historical_yahoo**(*ticker*, **date1**, **date2**, **asobject**=False, **adjusted**=True, **cachename**=None)

Get historical data for ticker between date1 and date2.
   
This function has been deprecated in 1.4 in favor of quotes_yahoo_historical_ochl, which
maintains the original argument order, or quotes_yahoo_historical_ohlc, which uses the open-
high-low-close order. This function will be removed in 1.5

See parse_yahoo_historical() for explanation of output formats and the **asobject** and **adjusted**
kwargs.

Parameters **ticker** : str
   
stock ticker

**date1** : sequence of form (year, month, day), datetime, or date
   
start date

**date2** : sequence of form (year, month, day), datetime, or date
   
end date

**cachename** : str or None
   
is the name of the local file cache. If None, will default to the md5 hash or the url (which incorporates the ticker and date range)
Examples

```python
>>> sp = f.quotes_historical_yahoo('^GSPC', d1, d2,
    asobject=True, adjusted=True)
>>> returns = (sp.open[1:] - sp.open[:-1])/sp.open[1:]
>>> [n,bins,patches] = hist(returns, 100)
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```

`matplotlib.finance.quotes_historical_yahoo_ochl(ticker, date1, date2, asobject=False, adjusted=True, cachename=None)`

Get historical data for ticker between date1 and date2.

See `parse_yahoo_historical()` for explanation of output formats and the `asobject` and `adjusted` kwargs.

**Parameters**
- **ticker**: str  
  stock ticker
- **date1**: sequence of form (year, month, day), datetime, or date  
  start date
- **date2**: sequence of form (year, month, day), datetime, or date  
  end date
- **cachename**: str or None  
  is the name of the local file cache. If None, will default to the md5 hash or the url (which incorporates the ticker and date range)

Examples

```python
>>> sp = f.quotes_historical_yahoo_ochl('^GSPC', d1, d2,
    asobject=True, adjusted=True)
>>> returns = (sp.open[1:] - sp.open[:-1])/sp.open[1:]
>>> [n,bins,patches] = hist(returns, 100)
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```

`matplotlib.finance.quotes_historical_yahoo_ohlc(ticker, date1, date2, asobject=False, adjusted=True, cachename=None)`

Get historical data for ticker between date1 and date2.

See `parse_yahoo_historical()` for explanation of output formats and the `asobject` and `adjusted` kwargs.

**Parameters**
- **ticker**: str  
  stock ticker
- **date1**: sequence of form (year, month, day), datetime, or date  
  start date
- **date2**: sequence of form (year, month, day), datetime, or date  
  end date
end date

**cachename** : str or None

is the name of the local file cache. If None, will default to the md5 hash or the url (which incorporates the ticker and date range)

**Examples**

```python
>>> sp = f.quotes_historical_yahoo_ohlc('^GSPC', d1, d2,
   asobject=True, adjusted=True)
>>> returns = (sp.open[1:] - sp.open[:-1])/sp.open[1:]
>>> [n,bins,patches] = hist(returns, 100)
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```

**matplotlib.finance.volume_overlay** *(ax, opens, closes, volumes, colorup=u'k', colordown=u'r', width=4, alpha=1.0)*

Add a volume overlay to the current axes. The opens and closes are used to determine the color of the bar. -1 is missing. If a value is missing on one it must be missing on all

**Parameters**

- **ax** : Axes
  - an Axes instance to plot to
- **opens** : sequence
  - a sequence of opens
- **closes** : sequence
  - a sequence of closes
- **volumes** : sequence
  - a sequence of volumes
- **width** : int
  - the bar width in points
- **colorup** : color
  - the color of the lines where close >= open
- **colordown** : color
  - the color of the lines where close < open
- **alpha** : float
  - bar transparency

**Returns**

- **ret** : barCollection
  - The barrCollection added to the axes

**matplotlib.finance.volume_overlay2** *(ax, closes, volumes, colorup=u'k', colordown=u'r', width=4, alpha=1.0)*

Add a volume overlay to the current axes. The closes are used to determine the color of the bar. -1 is missing. If a value is missing on one it must be missing on all

**nb:** first point is not displayed - it is used only for choosing the right color

**Parameters**

- **ax** : Axes
  - an Axes instance to plot to
- **closes** : sequence
  - a sequence of closes
volumes : sequence
    a sequence of volumes
width : int
    the bar width in points
colorup : color
    the color of the lines where close >= open
colordown : color
    the color of the lines where close < open
alpha : float
    bar transparency

Returns ret: barCollection
    The barCollection added to the axes

matplotlib.finance.volume_overlay3(ax, quotes, colorup=u'k', colordown=u'r', width=4, alpha=1.0)

Add a volume overlay to the current axes. quotes is a list of (d, open, high, low, close, volume) and close-open is used to determine the color of the bar

Parameters ax: Axes
    an Axes instance to plot to
quotes : sequence of (time, open, high, low, close, ...) sequences
    data to plot. time must be in float date format - see date2num
width : int
    the bar width in points
colorup : color
    the color of the lines where close1 >= close0
colordown : color
    the color of the lines where close1 < close0
alpha : float
    bar transparency

Returns ret: barCollection
    The barCollection added to the axes
51.1 matplotlib.font_manager

A module for finding, managing, and using fonts across platforms.

This module provides a single FontManager instance that can be shared across backends and platforms. The findfont() function returns the best TrueType (TTF) font file in the local or system font path that matches the specified FontProperties instance. The FontManager also handles Adobe Font Metrics (AFM) font files for use by the PostScript backend.

The design is based on the W3C Cascading Style Sheet, Level 1 (CSS1) font specification. Future versions may implement the Level 2 or 2.1 specifications.

Experimental support is included for using fontconfig on Unix variant platforms (Linux, OS X, Solaris). To enable it, set the constant USE_FONTCONFIG in this file to True. Fontconfig has the advantage that it is the standard way to look up fonts on X11 platforms, so if a font is installed, it is much more likely to be found.

```python
class matplotlib.font_manager.FontEntry(fname='', name='', style='normal', variant='normal', weight='normal', stretch='normal', size='medium')
Bases: object

A class for storing Font properties. It is used when populating the font lookup dictionary.

class matplotlib.font_manager.FontManager(size=None, weight='normal')
Bases: object

On import, the FontManager singleton instance creates a list of TrueType fonts based on the font properties: name, style, variant, weight, stretch, and size. The findfont() method does a nearest neighbor search to find the font that most closely matches the specification. If no good enough match is found, a default font is returned.

findfont(prop, fontext='ttf', directory=None, fallback_to_default=True, rebuild_if_missing=True)

Search the font list for the font that most closely matches the FontProperties prop.

findfont() performs a nearest neighbor search. Each font is given a similarity score to the target font properties. The first font with the highest score is returned. If no matches below a certain threshold are found, the default font (usually Vera Sans) is returned.
directory, is specified, will only return fonts from the given directory (or subdirectory of that directory).

The result is cached, so subsequent lookups don’t have to perform the O(n) nearest neighbor search.

If `fallback_to_default` is True, will fallback to the default font family (usually “Bitstream Vera Sans” or “Helvetica”) if the first lookup hard-fails.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

```python
static get_default_size()
    Return the default font size.

get_default_weight()
    Return the default font weight.

score_family(families, family2)
    Returns a match score between the list of font families in `families` and the font family name `family2`.
    An exact match anywhere in the list returns 0.0.
    A match by generic font name will return 0.1.
    No match will return 1.0.

score_size(size1, size2)
    Returns a match score between `size1` and `size2`.
    If `size2` (the size specified in the font file) is ‘scalable’, this function always returns 0.0, since any font size can be generated.
    Otherwise, the result is the absolute distance between `size1` and `size2`, normalized so that the usual range of font sizes (6pt - 72pt) will lie between 0.0 and 1.0.

score_stretch(stretch1, stretch2)
    Returns a match score between `stretch1` and `stretch2`.
    The result is the absolute value of the difference between the CSS numeric values of `stretch1` and `stretch2`, normalized between 0.0 and 1.0.

score_style(style1, style2)
    Returns a match score between `style1` and `style2`.
    An exact match returns 0.0.
    A match between ‘italic’ and ‘oblique’ returns 0.1.
    No match returns 1.0.

score_variant(variant1, variant2)
    Returns a match score between `variant1` and `variant2`.
    An exact match returns 0.0, otherwise 1.0.
score_weight(weight1, weight2)
   Returns a match score between weight1 and weight2.
   The result is the absolute value of the difference between the CSS numeric values of weight1 and weight2, normalized between 0.0 and 1.0.

set_default_weight(weight)
   Set the default font weight. The initial value is ‘normal’.

update_fonts(filenames)
   Update the font dictionary with new font files. Currently not implemented.

class matplotlib.font_manager.FontProperties(family=None, style=None, variant=None, weight=None, stretch=None, size=None, fname=None, _init=None)
   Bases: object
   A class for storing and manipulating font properties.
   The font properties are those described in the W3C Cascading Style Sheet, Level 1 font specification.
   The six properties are:
   • family: A list of font names in decreasing order of priority. The items may include a generic font family name, either ‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, or ‘monospace’. In that case, the actual font to be used will be looked up from the associated rcParam in matplotlibrc.
   • style: Either ‘normal’, ‘italic’ or ‘oblique’.
   • variant: Either ‘normal’ or ‘small-caps’.

   The default font property for TrueType fonts (as specified in the default matplotlibrc file) is:
   sans-serif, normal, normal, normal, normal, scalable.

   Alternatively, a font may be specified using an absolute path to a .ttf file, by using the fname kwarg.

   The preferred usage of font sizes is to use the relative values, e.g., ‘large’, instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager’s default font size.

   This class will also accept a fontconfig pattern, if it is the only argument provided. See the documentation on fontconfig patterns. This support does not require fontconfig to be installed. We are merely borrowing its pattern syntax for use here.

   Note that matplotlib’s internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in matplotlib than in other applications that use fontconfig.
copy()

Return a deep copy of self

get_family()

Return a list of font names that comprise the font family.

get_file()

Return the filename of the associated font.

get_fontconfig_pattern()

Get a fontconfig pattern suitable for looking up the font as specified with fontconfig’s fc-match utility.

See the documentation on fontconfig patterns.

This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

get_name()

Return the name of the font that best matches the font properties.

get_size()

Return the font size.

get_size_in_points()

get_slant()

Return the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

get_stretch()


get_style()

Return the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

get_variant()

Return the font variant. Values are: ‘normal’ or ‘small-caps’.

get_weight()


set_family(family)

Change the font family. May be either an alias (generic name is CSS parlance), such as: ‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, or ‘monospace’, a real font name or a list of real font names. Real font names are not supported when text.usetex is True.

set_file(file)

Set the filename of the fontfile to use. In this case, all other properties will be ignored.

set_fontconfig_pattern(pattern)

Set the properties by parsing a fontconfig pattern.
See the documentation on `fontconfig patterns`.

This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

**set_name** *(family)*

Change the font family. May be either an alias (generic name is CSS parlance), such as: ‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, or ‘monospace’, a real font name or a list of real font names. Real font names are not supported when `text.usetex` is `True`.

**set_size** *(size)*


**set_slant** *(style)*

Set the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

**set_stretch** *(stretch)*


**set_style** *(style)*

Set the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

**set_variant** *(variant)*

Set the font variant. Values are: ‘normal’ or ‘small-caps’.

**set_weight** *(weight)*


```
matplotlib.font_manager.OSXInstalledFonts(directories=None, fontext='ttf')
```

Get list of font files on OS X - ignores font suffix by default.

**class** `matplotlib.font_manager.TempCache`

```
Bases: object

A class to store temporary caches that are (a) not saved to disk and (b) invalidated whenever certain font-related rcParams—namely the family lookup lists—are changed or the font cache is reloaded. This avoids the expensive linear search through all fonts every time a font is looked up.

get(prop)

invalidating_rcparams = (u'font.serif', u'font.sans-serif', u'font.cursive', u'font.fantasy', u'font.monospace')

make_rcparams_key()`

```

**set**(prop, value)
Matplotlib, Release 1.4.3

matplotlib.font_manager.afmFontProperty(fontpath, font)
A function for populating a FontKey instance by extracting information from the AFM font file.

font is a class:AFM instance.

matplotlib.font_manager.createFontList(fontfiles, fontext=u'ttf')
A function to create a font lookup list. The default is to create a list of TrueType fonts. An AFM font list can optionally be created.

matplotlib.font_manager.findSystemFonts(fontpaths=None, fontext=u'ttf')
Search for fonts in the specified font paths. If no paths are given, will use a standard set of system paths, as well as the list of fonts tracked by fontconfig if fontconfig is installed and available. A list of TrueType fonts are returned by default with AFM fonts as an option.

matplotlib.font_manager.findfont(prop, **kw)

matplotlib.font_manager.get_fontconfig_fonts(fontext=u'ttf')
Grab a list of all the fonts that are being tracked by fontconfig by making a system call to fc-list. This is an easy way to grab all of the fonts the user wants to be made available to applications, without needing knowing where all of them reside.

matplotlib.font_manager.get_fontext_synonyms(fontext)
Return a list of file extensions that are synonyms for the given file extension fileext.

matplotlib.font_manager.is_opentype_cff_font(filename)
Returns True if the given font is a Postscript Compact Font Format Font embedded in an OpenType wrapper. Used by the PostScript and PDF backends that can not subset these fonts.

matplotlib.font_manager.list_fonts(directory, extensions)
Return a list of all fonts matching any of the extensions, possibly upper-cased, found recursively under the directory.

matplotlib.font_manager.pickle_dump(data, filename)
Equivalent to pickle.dump(data, open(filename, ‘w’)) but closes the file to prevent filehandle leakage.

matplotlib.font_manager.pickle_load(filename)
Equivalent to pickle.load(open(filename, ‘r’)) but closes the file to prevent filehandle leakage.

matplotlib.font_manager.ttfFontProperty(font)
A function for populating the FontKey by extracting information from the TrueType font file.

font is a FT2Font instance.

matplotlib.font_manager.ttfdict_to_fnames(d)
Flatten a ttfdict to all the filenames it contains.

matplotlib.font_manager.weight_as_number(weight)
Return the weight property as a numeric value. String values are converted to their corresponding numeric value.

matplotlib.font_manager.win32FontDirectory()
Return the user-specified font directory for Win32. This is looked up from the registry key:
\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Fonts

If the key is not found, $WINDIR/Fonts will be returned.

matplotlib.font_manager.win32InstalledFonts(directory=\None, fontext=‘.ttf’)

Search for fonts in the specified font directory, or use the system directories if none given. A list of TrueType font filenames are returned by default, or AFM fonts if $fontext == ‘afm’.

### 51.2 matplotlib.fontconfig_pattern

A module for parsing and generating fontconfig patterns.

See the [fontconfig pattern specification](https://www.freedesktop.org/wiki/Software/fontconfig/spec/) for more information.

**class** matplotlib.fontconfig_pattern.FontconfigPatternParser

A simple pyParsing-based parser for fontconfig-style patterns.

See the [fontconfig pattern specification](https://www.freedesktop.org/wiki/Software/fontconfig/spec/) for more information.

**parse(pattern)**

Parse the given fontconfig pattern and return a dictionary of key/value pairs useful for initializing a font_manager.FontProperties object.

**matplotlib.fontconfig_pattern.family_escape()**

sub(repl, string[, count = 0]) -> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

**matplotlib.fontconfig_pattern.family_unescape()**

sub(repl, string[, count = 0]) -> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

**matplotlib.fontconfig_pattern.generate_fontconfig_pattern(d)**

Given a dictionary of key/value pairs, generates a fontconfig pattern string.

**matplotlib.fontconfig_pattern.value_escape()**

sub(repl, string[, count = 0]) -> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

**matplotlib.fontconfig_pattern.value_unescape()**

sub(repl, string[, count = 0]) -> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.
52.1 matplotlib.gridspec

`gridspec` is a module which specifies the location of the subplot in the figure.

**GridSpec** specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

**SubplotSpec** specifies the location of the subplot in the given `GridSpec`.

```python
class matplotlib.gridspec.GridSpec(nrows, ncols, left=None, bottom=None, right=None, top=None, wspace=None, hspace=None, width_ratios=None, height_ratios=None)
```

Bases: `matplotlib.gridspec.GridSpecBase`

A class that specifies the geometry of the grid that a subplot will be placed. The location of grid is determined by similar way as the SubplotParams.

The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

**get_subplot_params**(fig=None)

return a dictionary of subplot layout parameters. The default parameters are from rcParams unless a figure attribute is set.

**locally_modified_subplot_params()**

**tight_layout**(fig, renderer=None, pad=1.08, h_pad=None, w_pad=None, rect=None)

Adjust subplot parameters to give specified padding.

Parameters:

- **pad** [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.
- **h_pad, w_pad** [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.
- **rect** [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is `(0, 0, 1, 1)`.
update(**kwargs)
    Update the current values. If any kwarg is None, default to the current value, if set, otherwise to rc.

**class matplotlib.gridspec.GridSpecBase(nrows, ncols, height_ratios=None, width_ratios=None)**

Bases: object

A base class of GridSpec that specifies the geometry of the grid that a subplot will be placed.

The number of rows and number of columns of the grid need to be set. Optionally, the ratio of heights and widths of rows and columns can be specified.

get_geometry()
    get the geometry of the grid, e.g., 2,3

get_grid_positions(fig)
    return lists of bottom and top position of rows, left and right positions of columns.

get_height_ratios()

get_subplot_params(fig=None)

get_width_ratios()

new_subplotspec(loc, rowspan=1, colspan=1)
    create and return a SubplotSpec instance.

set_height_ratios(height_ratios)

set_width_ratios(width_ratios)

**class matplotlib.gridspec.GridSpecFromSubplotSpec(nrows, ncols, subplot_spec, wspace=None, hspace=None, height_ratios=None, width_ratios=None)**

Bases: matplotlib.gridspec.GridSpecBase

GridSpec whose subplot layout parameters are inherited from the location specified by a given SubplotSpec.

The number of rows and number of columns of the grid need to be set. An instance of SubplotSpec is also needed to be set from which the layout parameters will be inherited. The wspace and hspace of the layout can be optionally specified or the default values (from the figure or rcParams) will be used.

get_subplot_params(fig=None)
    return a dictionary of subplot layout parameters.

get_topmost_subplotspec()
    get the topmost SubplotSpec instance associated with the subplot
class matplotlib.gridspec.SubplotSpec(gridspec, num1, num2=None)

Bases: object

specifies the location of the subplot in the given GridSpec.

The subplot will occupy the num1-th cell of the given gridspec. If num2 is provided, the subplot will span between num1-th cell and num2-th cell.

The index starts from 0.

get_geometry()
get the subplot geometry, e.g., 2,2,3. Unlike SuplorParams, index is 0-based

get_gridspec()

get_position(fig, return_all=False)
update the subplot position from fig.subplotpars

get_topmost_subplotspec()
get the topmost SubplotSpec instance associated with the subplot
53.1 matplotlib.image

The image module supports basic image loading, rescaling and display operations.

class matplotlib.image.AxesImage(ax, cmap=None, norm=None, interpolation=None, origin=None, extent=None, filternorm=1, filterrad=4.0, resample=False, **kwargs)

Bases: matplotlib.image._AxesImageBase

interpolation and cmap default to their rc settings

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

extent is data axes (left, right, bottom, top) for making image plots registered with data plots. Default is to label the pixel centers with the zero-based row and column indices.

Additional kwargs are matplotlib.artist properties

get_extent()

Get the image extent: left, right, bottom, top

make_image(magnification=1.0)

set_extent(extent)

extent is data axes (left, right, bottom, top) for making image plots

This updates ax.dataLim, and, if autoscaling, sets viewLim to tightly fit the image, regardless of dataLim. Autoscaling state is not changed, so following this with ax.autoscale_view will redo the autoscaling in accord with dataLim.

class matplotlib.image.BboxImage(bbox, cmap=None, norm=None, interpolation=None, origin=None, filternorm=1, filterrad=4.0, resample=False, interp_at_native=True, **kwargs)

Bases: matplotlib.image._AxesImageBase

The Image class whose size is determined by the given bbox.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

interp_at_native is a flag that determines whether or not interpolation should still be applied when the image is displayed at its native resolution. A common use case for this is when displaying an
image for annotational purposes; it is treated similarly to Photoshop (interpolation is only used when
displaying the image at non-native resolutions).

kwargs are an optional list of Artist keyword args

contains(mouseevent)
   Test whether the mouse event occured within the image.

draw(artist, renderer, *args, **kwargs)

get_size()
   Get the numrows, numcols of the input image

get_window_extent(renderer=None)

make_image(renderer, magnification=1.0)

class matplotlib.image.FigureImage(fig, cmap=None, norm=None, offsetx=0, offsety=0, origin=None, **kwargs)
   Bases: matplotlib.artist.Artist, matplotlib.cm.ScalarMappable
cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

kwargs are an optional list of Artist keyword args

contains(mouseevent)
   Test whether the mouse event occured within the image.

draw(artist, renderer, *args, **kwargs)

get_extent()
   Get the image extent: left, right, bottom, top

get_size()
   Get the numrows, numcols of the input image

make_image(magnification=1.0)

set_array(A)
   Deprecated; use set_data for consistency with other image types.

set_data(A)
   Set the image array.

write_png(fname)
   Write the image to png file with fname

zorder = 0

class matplotlib.image.NonUniformImage(ax, **kwargs)
   Bases: matplotlib.image.AxesImage
kwparams are identical to those for AxesImage, except that ‘interpolation’ defaults to ‘nearest’, and
‘bilinear’ is the only alternative.

get_extent()

make_image(magnification=1.0)

set_array(*args)

set_cmap(cmap)

set_data(x, y, A)
   Set the grid for the pixel centers, and the pixel values.
   x and y are 1-D ndarrays of lengths N and M, respectively, specifying pixel centers
   A is an (M,N) ndarray or masked array of values to be colormapped, or a (M,N,3) RGB array, or a (M,N,4) RGBA array.

set_filternorm(s)

set_filterrad(s)

set_interpolation(s)

set_norm(norm)

class matplotlib.image.PcolorImage(ax, x=None, y=None, A=None, cmap=None, norm=None, **kwargs)
Bases: matplotlib.artist.Artist, matplotlib.cm.ScalarMappable
Make a pcolor-style plot with an irregular rectangular grid.
This uses a variation of the original irregular image code, and it is used by pcolorfast for the correspoding grid type.
cmap defaults to its rc setting
cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1
Additional kwparams are matplotlib.artist properties
changed()

draw(artist, renderer, *args, **kwargs)

make_image(magnification=1.0)
**set_alpha(alpha)**
Set the alpha value used for blending - not supported on all backends

ACCEPETS: float

**set_array(*args)**

**set_data(x, y, A)**

**matplotlib.image.from_images()**

**matplotlib.image.fromarray()**

**matplotlib.image.fromarray2()**

**matplotlib.image.frombuffer()**

**matplotlib.image.frombyte()**

**matplotlib.image.imread(fname, format=None)**
Read an image from a file into an array.

*fname* may be a string path or a Python file-like object. If using a file object, it must be opened in binary mode.

If *format* is provided, will try to read file of that type, otherwise the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Return value is a numpy.array. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.

matplotlib can only read PNGs natively, but if PIL is installed, it will use it to load the image and return an array (if possible) which can be used with imshow().

**matplotlib.image.imsave(fname, arr, vmin=None, vmax=None, cmap=None, format=None, origin=None, dpi=100)**
Save an array as in image file.

The output formats available depend on the backend being used.

**Arguments:**

*fname*: A string containing a path to a filename, or a Python file-like object. If *format* is None and *fname* is a string, the output format is deduced from the extension of the filename.

*arr*: An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

**Keyword arguments:**

*vmin/vmax*: [None | scalar] *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.
**cmap**: `cmap` is a `colors.Colormap` instance, e.g., `cm.jet`. If None, default to the rc `image.cmap` value.

**format**: One of the file extensions supported by the active backend. Most backends support `png`, `pdf`, `ps`, `eps` and `svg`.

**origin**: [ ‘upper’ | ‘lower’ ] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc `image.origin` value.

**dpi** The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

```python
matplotlib.image.pcolor()
```

```python
matplotlib.image.pcolor2()
```

```python
matplotlib.image.pil_to_array(pilImage)
```
Load a PIL image and return it as a numpy array. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4

```python
matplotlib.image.thumbnail(infile, thumbfile, scale=0.1, interpolation=u'bilinear', preview=False)
```
make a thumbnail of image in `infile` with output filename `thumbfile`.

- **infile** the image file – must be PNG or PIL readable if you have PIL installed
- **thumbfile** the thumbnail filename
- **scale** the scale factor for the thumbnail
- **interpolation** the interpolation scheme used in the resampling
- **preview** if True, the default backend (presumably a user interface backend) will be used which will cause a figure to be raised if `show()` is called. If it is False, a pure image backend will be used depending on the extension, ‘png’->FigureCanvasAgg, ‘pdf’->FigureCanvasPdf, ‘svg’->FigureCanvasSVG

See examples/misc/image_thumbnail.py.

Return value is the figure instance containing the thumbnail
54.1 matplotlib.legend

The legend module defines the Legend class, which is responsible for drawing legends associated with axes and/or figures.

**Important:** It is unlikely that you would ever create a Legend instance manually. Most users would normally create a legend via the `legend()` function. For more details on legends there is also a legend guide.

The Legend class can be considered as a container of legend handles and legend texts. Creation of corresponding legend handles from the plot elements in the axes or figures (e.g., lines, patches, etc.) are specified by the handler map, which defines the mapping between the plot elements and the legend handlers to be used (the default legend handlers are defined in the `legend_handler` module). Note that not all kinds of artist are supported by the legend yet by default but it is possible to extend the legend handler’s capabilities to support arbitrary objects. See the legend guide for more information.

```python
class matplotlib.legend.DraggableLegend(legend, use_blit=False, update=u’loc’)
    Bases: matplotlib.offsetbox.DraggableOffsetBox

    update : If “loc”, update loc parameter of legend upon finalizing. If “bbox”, update bbox_to_anchor parameter.

    artist_picker(legend, evt)

    finalize_offset()
```

```python
class matplotlib.legend.Legend(parent, handles, labels, loc=None, numpoints=None, markerscale=None, scatterpoints=None, scatteryoffsets=None, prop=None, fontsize=None, borderpad=None, labelspacing=None, handlelength=None, handleheight=None, handletextpad=None, borderaxespad=None, columnspacing=None, ncol=1, mode=None, fancybox=None, shadow=None, title=None, framealpha=None, bbox_to_anchor=None, bbox_transform=None, frameon=None, handler_map=None)
    Bases: matplotlib.artist.Artist
```
Place a legend on the axes at location loc. Labels are a sequence of strings and loc can be a string or an integer specifying the legend location.

The location codes are:

- 'best' : 0, (only implemented for axis legends)
- 'upper right' : 1,
- 'upper left' : 2,
- 'lower left' : 3,
- 'lower right' : 4,
- 'right' : 5,
- 'center left' : 6,
- 'center right' : 7,
- 'lower center' : 8,
- 'upper center' : 9,
- 'center' : 10,

loc can be a tuple of the normalized coordinate values with respect its parent.

- *parent*: the artist that contains the legend
- *handles*: a list of artists (lines, patches) to be added to the legend
- *labels*: a list of strings to label the legend

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>a location code</td>
</tr>
<tr>
<td>prop</td>
<td>the font property</td>
</tr>
<tr>
<td>fontsize</td>
<td>the font size (used only if prop is not specified)</td>
</tr>
<tr>
<td>markerscale</td>
<td>the relative size of legend markers vs. original</td>
</tr>
<tr>
<td>numpoints</td>
<td>the number of points in the legend for line</td>
</tr>
<tr>
<td>scatterpoints</td>
<td>the number of points in the legend for scatter plot</td>
</tr>
<tr>
<td>scatteryoffsets</td>
<td>a list of yoffsets for scatter symbols in legend</td>
</tr>
<tr>
<td>frameon</td>
<td>if True, draw a frame around the legend. If None, use rc</td>
</tr>
<tr>
<td>fancybox</td>
<td>if True, draw a frame with a round fancybox. If None, use rc</td>
</tr>
<tr>
<td>shadow</td>
<td>if True, draw a shadow behind legend</td>
</tr>
<tr>
<td>framealpha</td>
<td>If not None, alpha channel for the frame.</td>
</tr>
<tr>
<td>ncol</td>
<td>number of columns</td>
</tr>
<tr>
<td>borderpad</td>
<td>the fractional whitespace inside the legend border</td>
</tr>
<tr>
<td>labelspacing</td>
<td>the vertical space between the legend entries</td>
</tr>
<tr>
<td>handlelength</td>
<td>the length of the legend handles</td>
</tr>
<tr>
<td>handleheight</td>
<td>the height of the legend handles</td>
</tr>
<tr>
<td>handletextpad</td>
<td>the pad between the legend handle and text</td>
</tr>
<tr>
<td>borderaxespad</td>
<td>the pad between the axes and legend border</td>
</tr>
<tr>
<td>columnspacing</td>
<td>the spacing between columns</td>
</tr>
<tr>
<td>title</td>
<td>the legend title</td>
</tr>
<tr>
<td>bbox_to_anchor</td>
<td>the bbox that the legend will be anchored.</td>
</tr>
<tr>
<td>bbox_transform</td>
<td>the transform for the bbox. transAxes if None.</td>
</tr>
</tbody>
</table>

The pad and spacing parameters are measured in font-size units. e.g., a fontsize of 10 points and a handlelength=5 implies a handlelength of 50 points. Values from rcParams will be used if None.

Users can specify any arbitrary location for the legend using the *bbox_to_anchor* keyword argument.
bbox_to_anchor can be an instance of BboxBase (or its derivatives) or a tuple of 2 or 4 floats. See set_bbox_to_anchor() for more detail.

The legend location can be specified by setting loc with a tuple of 2 floats, which is interpreted as the lower-left corner of the legend in the normalized axes coordinate.

codes = {u'right': 5, u'center left': 6, u'upper right': 1, u'lower right': 4, u'best': 0, u'center': 10, u'lower left': 3, u'center right': 7, u'upper left': 2, u'upper center': 9, u'lower center': 8}

contains(event)

draggable(state=None, use_blit=False, update=u'loc')
    Set the draggable state – if state is
        • None : toggle the current state
        • True : turn draggable on
        • False : turn draggable off
    If draggable is on, you can drag the legend on the canvas with the mouse. The DraggableLegend helper instance is returned if draggable is on.

    The update parameter control which parameter of the legend changes when dragged. If update is “loc”, the loc paramter of the legend is changed. If “bbox”, the bbox_to_anchor parameter is changed.

draw(artist, renderer, *args, **kwargs)
    Draw everything that belongs to the legend

draw_frame(b)
    b is a boolean. Set draw frame to b

get_bbox_to_anchor()
    return the bbox that the legend will be anchored

get_children()
    return a list of child artists

classmethod get_default_handler_map()
    A class method that returns the default handler map.

get_frame()
    return the Rectangle instance used to frame the legend

get_frame_on()
    Get whether the legend box patch is drawn

static get_legend_handler(legend_handler_map, orig_handle)
    return a legend handler from legend_handler_map that corresponds to orig_handle.

    legend_handler_map should be a dictionary object (that is returned by the get_legend_handler_map method).

    It first checks if the orig_handle itself is a key in the legend_handler_map and return the associated value. Otherwise, it checks for each of the classes in its method-resolution-order. If no matching key is found, it returns None.
get_legend_handler_map()  
return the handler map.

get_lines()  
return a list of lines.Line2D instances in the legend

get_patches()  
return a list of patch instances in the legend

get_texts()  
return a list of text.Text instance in the legend

get_title()  
return Text instance for the legend title

get_window_extent(*args, **kwargs)  
return a extent of the the legend

set_bbox_to_anchor(bbox, transform=None)  
set the bbox that the legend will be anchored.  
  
  bbox can be a BboxBase instance, a tuple of [left, bottom, width, height] in the given transform  
  (normalized axes coordinate if None), or a tuple of [left, bottom] where the width and height  
  will be assumed to be zero.

classmethod set_default_handler_map(handler_map)  
A class method to set the default handler map.

set_frame_on(b)  
Set whether the legend box patch is drawn  
  ACCEPTS: [ True | False ]

set_title(title, prop=None)  
set the legend title. Fontproperties can be optionally set with prop parameter.

classmethod update_default_handler_map(handler_map)  
A class method to update the default handler map.

zorder = 5

54.2 matplotlib.legend_handler

This module defines default legend handlers.
It is strongly encouraged to have read the legend guide before this documentation.
Legend handlers are expected to be a callable object with a following signature.

legend_handler(legend, orig_handle, fontsize, handlebox)
Where `legend` is the legend itself, `orig_handle` is the original plot, `fontsize` is the fontsize in pixels, and `handlebox` is a OffsetBox instance. Within the call, you should create relevant artists (using relevant properties from the `legend` and/or `orig_handle`) and add them into the handlebox. The artists needs to be scaled according to the fontsize (note that the size is in pixel, i.e., this is dpi-scaled value).

This module includes definition of several legend handler classes derived from the base class (HandlerBase) with the following method.

```python
def legend_artist(self, legend, orig_handle, fontsize, handlebox):
```

**class matplotlib.legend_handler.HandlerBase(xpad=0.0, ypad=0.0, update_func=None)**

A Base class for default legend handlers.

The derived classes are meant to override `create_artists` method, which has a following signature:

```python
def create_artists(self, legend, orig_handle,
                  xdescent, ydescent, width, height, fontsize,
                  trans):
```

The overridden method needs to create artists of the given transform that fits in the given dimension (xdescent, ydescent, width, height) that are scaled by fontsize if necessary.

```python
adjuset_drawing_area(legend, orig_handle, xdescent, ydescent, width, height, fontsize)
```

```python
create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)
```

```python
legend_artist(legend, orig_handle, fontsize, handlebox)
```

Return the artist that this HandlerBase generates for the given original artist/handle.

**Parameters**

- `legend` : matplotlib.legend.Legend instance
  The legend for which these legend artists are being created.
- `orig_handle` : matplotlib.artist.Artist or similar
  The object for which these legend artists are being created.
- `fontsize` : float or int
  The fontsize in pixels. The artists being created should be scaled according to the given fontsize.
- `handlebox` : matplotlib.offsetbox.OffsetBox instance
  The box which has been created to hold this legend entry’s artists. Artists created in the `legend_artist` method must be added to this handlebox inside this method.

```python
update_prop(legend_handle, orig_handle, legend)
```

**class matplotlib.legend_handler.HandlerCircleCollection(yoffsets=None, sizes=None, **kw)**

Handler for CircleCollections

```python
create_collection(orig_handle, sizes, offsets, transOffset)
```
class matplotlib.legend_handler.HandlerErrorbar(xerr_size=0.5, yerr_size=None, marker_pad=0.3, numpoints=None, **kw)

Handler for Errorbars

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

get_err_size(legend, xdescent, ydescent, width, height, fontsize)

class matplotlib.legend_handler.HandlerLine2D(marker_pad=0.3, numpoints=None, **kw)

Handler for Line2D instances.

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

class matplotlib.legend_handler.HandlerLineCollection(marker_pad=0.3, numpoints=None, **kw)

Handler for LineCollection instances.

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

get_numpoints(legend)

class matplotlib.legend_handler.HandlerNpoints(marker_pad=0.3, numpoints=None, **kw)

get_numpoints(legend)

get_xdata(legend, xdescent, ydescent, width, height, fontsize)

class matplotlib.legend_handler.HandlerNpointsYoffsets(numpoints=None, yoffsets=None, **kw)

get_ydata(legend, xdescent, ydescent, width, height, fontsize)

class matplotlib.legend_handler.HandlerPatch(patch_func=None, **kw)

Handler for Patch instances.

The HandlerPatch class optionally takes a function patch_func who's responsibility is to create the legend key artist. The patch_func should have the signature:

def patch_func(legend=legend, orig_handle=orig_handle, xdescent=xdescent, ydescent=ydescent, width=width, height=height, fontsize=fontsize)

Subsequently the created artist will have its update_prop method called and the appropriate transform will be applied.
create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

class matplotlib.legend_handler.HandlerPathCollection(yoffsets=None, sizes=None, **kw)
    Handler for PathCollections, which are used by scatter

create_collection(orig_handle, sizes, offsets, transOffset)

class matplotlib.legend_handler.HandlerRegularPolyCollection(yoffsets=None, sizes=None, **kw)
    Handler for RegularPolyCollections.

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

create_collection(orig_handle, sizes, offsets, transOffset)

get_numpoints(legend)

get_sizes(legend, orig_handle, xdescent, ydescent, width, height, fontsize)

update_prop(legend_handle, orig_handle, legend)

class matplotlib.legend_handler.HandlerStem(marker_pad=0.3, numpoints=None, bottom=None, yoffsets=None, **kw)
    Handler for Errorbars

class matplotlib.legend_handler.HandlerTuple(**kwargs)
    Handler for Tuple

matplotlib.legend_handler.update_from_first_child(tgt, src)
55.1 `matplotlib.lines`  

This module contains all the 2D line class which can draw with a variety of line styles, markers and colors.

```python
class matplotlib.lines.Line2D(xdata, ydata, linewidth=None, linestyle=None, color=None, marker=None, markersize=None, markeredgewidth=None, markeredgecolor=None, markerfacecolor=None, markerfacecoloralt=u'none', fillstyle=u'full', antialiased=None, dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None, pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

Bases: `matplotlib.artist.Artist`

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create “stepped” lines in various styles.

Create a `Line2D` instance with `x` and `y` data in sequences `xdata`, `ydata`.

The kwargs are `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘but’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
</tbody>
</table>
Table 55.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See `set_linestyle()` for a description of the line styles, `set_marker()` for a description of the markers, and `set_drawstyle()` for a description of the draw styles.

`contains(mouseevent)`

Test whether the mouse event occurred on the line. The pick radius determines the precision of the location test (usually within five points of the value). Use `get_pickradius()` or `set_pickradius()` to view or modify it.

Returns `True` if any values are within the radius along with `{'ind': pointlist}`, where `pointlist` is the set of points within the radius.

TODO: sort returned indices by distance

`draw(artist, renderer, *args, **kwargs)`

draw the Line with `renderer` unless visibility is False

`drawStyleKeys = [u'default', u'steps-mid', u'steps-pre', u'steps-post', u'steps']`
drawStyles = {u'default': u'_draw_lines', u'steps-mid': u'_draw_steps_mid', u'steps': u'_draw_steps_pre', u'steps-pre': u'_draw_steps_pre', u'steps-post': u'_draw_steps_post'}

fillStyles = (u'full', u'left', u'right', u'bottom', u'top', u'none')

filled_markers = (u'o', u'v', u'^', u'<', u'>', u'8', u's', u'p', u'^*', u'v', u'H', u'D', u'd')

get_aa()
    alias for get_antialiased

get_antialiased()

get_c()
    alias for get_color

get_color()

get_dash_capstyle()
    Get the cap style for dashed linestyles

get_dash_joinstyle()
    Get the join style for dashed linestyles

get_data(orig=True)
    Return the xdata, ydata.
    If orig is True, return the original data.

get_drawstyle()

get_fillstyle()
    return the marker fillstyle

get_linestyle()

get_linewidth()

get_ls()
    alias for get_linestyle

get_lw()
    alias for get_linewidth

get_marker()

get_markeredgecolor()}
get_markeredgewidth()

get_markerfacecolor()

get_markerfacecoloralt()

get_markersize()

get_markevery()
    return the markevery setting

get_mec()
    alias for get_markeredgewidth

get_mew()
    alias for get_markeredgewidth

get_mfc()
    alias for get_markerfacecolor

get_mfcalt(alt=False)
    alias for get_markerfacecoloralt

get_ms()
    alias for get_markersize

get_path()
    Return the Path object associated with this line.

get_pickradius()
    return the pick radius used for containment tests

get_solid_capstyle()
    Get the cap style for solid linestyles

get_solid_joinstyle()
    Get the join style for solid linestyles

get_window_extent(renderer)

get_xdata(orig=True)
    Return the xdata.
    If orig is True, return the original data, else the processed data.

get_xydata()
    Return the xy data as a Nx2 numpy array.

get_ydata(orig=True)
    Return the ydata.
    If orig is True, return the original data, else the processed data.
is_dashed()
    return True if line is dashstyle

lineStyles = {u'': u'_draw_nothing', u' ': u'_draw_nothing', u'None': u'_draw_nothing', u'–': u'_draw_dashed', u'-.': u'_draw_dash_dot', u'-': u'_draw_solid', u':': u'_draw_dotted'}

markers = {0: u'tickleft', 1: u'tickright', 2: u'tickup', 3: u'tickdown', 4: u'caretlefl', u'D': u'diamond', 6: u'caretup', 7: u'caretdown', u's': u'square', u'|': u'vline', u'': u'nothing', u'None': u'nothing', None: u'nothing', u'x': u'x', 5: u'caretright', u'_': u'hline', u'^': u'triangle_up', u' ': u'nothing', u'd': u'thin_diamond', u'h': u'hexagon1', u'+': u'plus', u'*': u'star', u'.,': u'pixel', u'o': u'circle', u'.': u'point', u'1': u'tri_down', u'p': u'pentagon', u'3': u'tri_left', u'2': u'tri_up', u'4': u'tri_right', u'H': u'hexagon2', u'v': u'triangle_down', u'8': u'octagon', u'<': u'triangle_left', u'>': u'triangle_right'}

recache(always=False)

recache_always()

set_aa(val)
    alias for set_antialiased

set_antialiased(b)
    True if line should be drawin with antialiased rendering
    ACCEPTS: [True | False]

set_axes(ax)
    Set the Axes instance in which the artist resides, if any.
    ACCEPTS: an Axes instance

set_c(val)
    alias for set_color

set_color(color)
    Set the color of the line
    ACCEPTS: any matplotlib color

set_dash_capstyle(s)
    Set the cap style for dashed linestyles
    ACCEPTS: [‘butt’ | ‘round’ | ‘projecting’]

set_dash_joinstyle(s)
    Set the join style for dashed linestyles ACCEPTS: [‘miter’ | ‘round’ | ‘bevel’]

set_dashes(seq)
    Set the dash sequence, sequence of dashes with on off ink in points. If seq is empty or if seq =
    (None, None), the linestyle will be set to solid.
    ACCEPTS: sequence of on/off ink in points

set_data(*args)
    Set the x and y data
    ACCEPTS: 2D array (rows are x, y) or two 1D arrays

set_drawstyle(drawstyle)
    Set the drawstyle of the plot
‘default’ connects the points with lines. The steps variants produce step-plots. ‘steps’ is equi-
valent to ‘steps-pre’ and is maintained for backward-compatibility.

**ACCEPTS:** ['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']

```
defaultset_fillstyle(fs)
Set the marker fill style; ‘full’ means fill the whole marker. ‘none’ means no filling; other
options are for half-filled markers.

**ACCEPTS:** ['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
```

```
defaultset_linestyle(linestyle)
Set the linestyle of the line (also accepts drawstyles)

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>solid</td>
</tr>
<tr>
<td>'--'</td>
<td>dashed</td>
</tr>
<tr>
<td>'-.'</td>
<td>dash_dot</td>
</tr>
<tr>
<td>':'</td>
<td>dotted</td>
</tr>
<tr>
<td>'None'</td>
<td>draw nothing</td>
</tr>
<tr>
<td>''</td>
<td>draw nothing</td>
</tr>
<tr>
<td></td>
<td>draw nothing</td>
</tr>
</tbody>
</table>

‘steps’ is equivalent to ‘steps-pre’ and is maintained for backward-compatibility.

See also:

```
defaultset_drawstyle() To set the drawing style (stepping) of the plot.

**ACCEPTS:** ['-' | '--' | '-' | ':' | 'None' | ' ' | ' ']
```

and any drawstyle in combination with a linestyle, e.g., 'steps--'.

```
defaultset_linewidth(w)
Set the line width in points

**ACCEPTS:** float value in points
```

```
defaultset_ls(val)
alias for set_linestyle
```

```
defaultset_lw(val)
alias for set_linewidth
```

```
defaultset_marker(marker)
Set the line marker

**ACCEPTS:** A valid marker style

**Parameters** marker: marker style

See markers for full description of possible argument
```

```
defaultset_markeredgecolor(ec)
Set the marker edge color

**ACCEPTS:** any matplotlib color
```

```
defaultset_markeredgewidth(ew)
Set the marker edge width in points
```

1256 Chapter 55. lines
ACCEPTE: float value in points

**set_markerfacecolor**(fc)
Set the marker face color.
ACCEPTE: any matplotlib color

**set_markerfacecoloralt**(fc)
Set the alternate marker face color.
ACCEPTE: any matplotlib color

**set_markersize**(sz)
Set the marker size in points
ACCEPTE: float

**set_markevery**(every)
Set the markevery property to subsample the plot when using markers.
e.g., if `every`=5, every 5-th marker will be plotted.
ACCEPTE: [None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]

**Parameters**

- **every**: None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float

Which markers to plot.
- `every`=None, every point will be plotted.
- `every`=N, every N-th marker will be plotted starting with marker 0.
- `every`=(start, N), every N-th marker, starting at point start, will be plotted.
- `every`=slice(start, end, N), every N-th marker, starting at point start, upto but not including point end, will be plotted.
- `every`=[i, j, m, n], only markers at points i, j, m, and n will be plotted.
- `every`=0.1, (i.e. a float) then markers will be spaced at approximately equal distances along the line; the distance along the line between markers is determined by multiplying the display-coordinate distance of the axes bounding-box diagonal by the value of every.
- `every`=(0.5, 0.1) (i.e. a length-2 tuple of float), the same functionality as `every`=0.1 is exhibited but the first marker will be 0.5 multiplied by the display-coordinate-diagonal-distance along the line.

**Notes**

Setting the markevery property will only show markers at actual data points. When using float arguments to set the markevery property on irregularly spaced data, the markers will likely not
appear evenly spaced because the actual data points do not coincide with the theoretical spacing between markers.

When using a start offset to specify the first marker, the offset will be from the first data point which may be different from the first the visible data point if the plot is zoomed in.

If zooming in on a plot when using float arguments then the actual data points that have markers will change because the distance between markers is always determined from the display-coordinates axes-bounding-box-diagonal regardless of the actual axes data limits.

```python
set_mec(val)
    alias for set_markeredgecolor

set_mew(val)
    alias for set_markeredgewidth

set_mfc(val)
    alias for set_markerfacecolor

set_mfcalt(val)
    alias for set_markerfacecoloralt

set_ms(val)
    alias for set_markersize

set_picker(p)
    Sets the event picker details for the line.
    ACCEPTS: float distance in points or callable pick function fn(artist, event)

set_pickradius(d)
    Sets the pick radius used for containment tests
    ACCEPTS: float distance in points

set_solid_capstyle(s)
    Set the cap style for solid linestyles
    ACCEPTS: ['butt' | 'round' | 'projecting']

set_solid_joinstyle(s)
    Set the join style for solid linestyles ACCEPTS: ['miter' | 'round' | 'bevel']

set_transform(t)
    set the Transformation instance used by this artist
    ACCEPTS: a matplotlib.transforms.Transform instance

set_xdata(x)
    Set the data np.array for x
    ACCEPTS: 1D array

set_ydata(y)
    Set the data np.array for y
    ACCEPTS: 1D array
update_from(other)
    copy properties from other to self

validCap = (u'butt', u'round', u'projecting')

validJoin = (u'miter', u'round', u'bevel')

zorder = 2

class matplotlib.lines.VertexSelector(line)
    Manage the callbacks to maintain a list of selected vertices for matplotlib.lines.Line2D. Derived classes should override process_selected() to do something with the picks.

    Here is an example which highlights the selected verts with red circles:

        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.lines as lines

        class HighlightSelected(lines.VertexSelector):
            def __init__(self, line, fmt='ro', **kwargs):
                lines.VertexSelector.__init__(self, line)
                self.markers, = self.axes.plot([], [], fmt, **kwargs)

            def process_selected(self, ind, xs, ys):
                self.markers.set_data(xs, ys)
                self.canvas.draw()

        fig = plt.figure()
        ax = fig.add_subplot(111)
        x, y = np.random.rand(2, 30)
        line, = ax.plot(x, y, 'bs-', picker=5)
        selector = HighlightSelected(line)
        plt.show()

    Initialize the class with a matplotlib.lines.Line2D instance. The line should already be added to some matplotlib.axes.Axes instance and should have the picker property set.

onpick(event)
    When the line is picked, update the set of selected indicies.

process_selected(ind, xs, ys)
    Default “do nothing” implementation of the process_selected() method.

        ind are the indices of the selected vertices. xs and ys are the coordinates of the selected vertices.

matplotlib.lines.segment_hits(cx, cy, x, y, radius)
    Determine if any line segments are within radius of a point. Returns the list of line segments that are within that radius.
### 56.1 matplotlib.markers

This module contains functions to handle markers. Used by both the marker functionality of `plot` and `scatter`.

All possible markers are defined here:

<table>
<thead>
<tr>
<th>marker</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;.&quot;</td>
<td>point</td>
</tr>
<tr>
<td>&quot;,&quot;</td>
<td>pixel</td>
</tr>
<tr>
<td>&quot;o&quot;</td>
<td>circle</td>
</tr>
<tr>
<td>&quot;v&quot;</td>
<td>triangle_down</td>
</tr>
<tr>
<td>&quot;^&quot;</td>
<td>triangle_up</td>
</tr>
<tr>
<td>&quot;&lt;&quot;</td>
<td>triangle_left</td>
</tr>
<tr>
<td>&quot;&gt;&quot;</td>
<td>triangle_right</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>tri_down</td>
</tr>
<tr>
<td>&quot;2&quot;</td>
<td>tri_up</td>
</tr>
<tr>
<td>&quot;3&quot;</td>
<td>tri_left</td>
</tr>
<tr>
<td>&quot;4&quot;</td>
<td>tri_right</td>
</tr>
<tr>
<td>&quot;8&quot;</td>
<td>octagon</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>square</td>
</tr>
<tr>
<td>&quot;p&quot;</td>
<td>pentagon</td>
</tr>
<tr>
<td>&quot;*&quot;</td>
<td>star</td>
</tr>
<tr>
<td>&quot;h&quot;</td>
<td>hexagon1</td>
</tr>
<tr>
<td>&quot;H&quot;</td>
<td>hexagon2</td>
</tr>
<tr>
<td>&quot;+&quot;</td>
<td>plus</td>
</tr>
<tr>
<td>&quot;x&quot;</td>
<td>x</td>
</tr>
<tr>
<td>&quot;D&quot;</td>
<td>diamond</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>thin_diamond</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;:&quot;</td>
<td>hline</td>
</tr>
<tr>
<td>TICKLEFT</td>
<td>tickleft</td>
</tr>
<tr>
<td>TICKRIGHT</td>
<td>tickright</td>
</tr>
<tr>
<td>TICKUP</td>
<td>tickup</td>
</tr>
<tr>
<td>TICKDOWN</td>
<td>tickdown</td>
</tr>
</tbody>
</table>
Table 56.1 – continued from previous page

<table>
<thead>
<tr>
<th>marker</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARETLEFT</td>
<td>caretleft</td>
</tr>
<tr>
<td>CARETRIGHT</td>
<td>caretright</td>
</tr>
<tr>
<td>CARETUP</td>
<td>caretpup</td>
</tr>
<tr>
<td>CARETDOWN</td>
<td>caretdown</td>
</tr>
<tr>
<td>“None”</td>
<td>nothing</td>
</tr>
<tr>
<td>None</td>
<td>nothing</td>
</tr>
<tr>
<td>“”</td>
<td>nothing</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>nothing</td>
</tr>
<tr>
<td>'...'</td>
<td>render the string using mathtext.</td>
</tr>
<tr>
<td>verts</td>
<td>a list of (x, y) pairs used for Path vertices. The center of the marker is located at (0,0) and the size is normalized.</td>
</tr>
<tr>
<td>path</td>
<td>a Path instance.</td>
</tr>
<tr>
<td>(numsides, style, angle)</td>
<td>see below</td>
</tr>
</tbody>
</table>

The marker can also be a tuple (numsides, style, angle), which will create a custom, regular symbol.

- **numsides**: the number of sides
- **style**: the style of the regular symbol:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a regular polygon</td>
</tr>
<tr>
<td>1</td>
<td>a star-like symbol</td>
</tr>
<tr>
<td>2</td>
<td>an asterisk</td>
</tr>
<tr>
<td>3</td>
<td>a circle (numsides and angle is ignored)</td>
</tr>
</tbody>
</table>

- **angle**: the angle of rotation of the symbol, in degrees

For backward compatibility, the form (verts, 0) is also accepted, but it is equivalent to just verts for giving a raw set of vertices that define the shape.

```python
class matplotlib.markers.MarkerStyle(marker=None, fillstyle='full')
```

Bases: object

MarkerStyle

- **Parameters**
  - **marker**: string or array_like, optional, default: None
    - See the descriptions of possible markers in the module docstring.
  - **fillstyle**: string, optional, default: ‘full’

**Attributes**

- **markers**: (list of known markes)
- **fillstyles** : (list of known fillstyles)
- **filled_markers**: (list of known filled markers.)

```python
filled_markers = (u'o', u'v', u'^', u'<', u'>', u'8', u's', u'p', u'*', u'h', u'H', u'D', u'd')
```
fillstyles = (u'full', u'left', u'right', u'bottom', u'top', u'none')

get_alt_path()

get_alt_transform()

get_capstyle()

get_fillstyle()

get_joinstyle()

get_marker()

get_path()

get_snap_threshold()

get_transform()

is_filled()

markers = {0: u'tickleft', 1: u'tickright', 2: u'tickup', 3: u'tickdown', 4: u'caretleft', u'D': u'diamond', 6: u'caretup', 7: u'caretdown', u's': u'square', u'|': u'vline', u'': u'nothing', u'None': u'nothing', None: u'nothing', u'x': u'x', 5: u'caretright', u'_': u'hline', u'^': u'triangle_up', u' ': u'nothing', u'd': u'thin_diamond', u'h': u'hexagon1', u'+': u'plus', u'*': u'star', u',': u'pixel', u'o': u'circle', u'.': u'point', u'1': u'tri_down', u'p': u'pentagon', u'3': u'tri_left', u'2': u'tri_up', u'4': u'tri_right', u'H': u'hexagon2', u'v': u'triangle_down', u'8': u'octagon', u'<': u'triangle_left', u'>': u'triangle_right'}

set_fillstyle(fillstyle)
    Sets fillstyle
    Parameters fillstyle : string amongst known fillstyles

set_marker(marker)
57.1 matplotlib.mathtext

mathtext is a module for parsing a subset of the TeX math syntax and drawing them to a matplotlib backend.

For a tutorial of its usage see Writing mathematical expressions. This document is primarily concerned with implementation details.

The module uses pyparsing to parse the TeX expression.

The Bakoma distribution of the TeX Computer Modern fonts, and STIX fonts are supported. There is experimental support for using arbitrary fonts, but results may vary without proper tweaking and metrics for those fonts.

If you find TeX expressions that don’t parse or render properly, please email mdroe@stsci.edu, but please check KNOWN ISSUES below first.

class matplotlib.mathtext.Accent(c, state)
   Bases: matplotlib.mathtext.Char

   The font metrics need to be dealt with differently for accents, since they are already offset correctly from the baseline in TrueType fonts.

   grow()

   render(x, y)
       Render the character to the canvas.

   shrink()

class matplotlib.mathtext.AutoHeightChar(c, height, depth, state, always=False, factor=None)
   Bases: matplotlib.mathtext.Hlist

   AutoHeightChar will create a character as close to the given height and depth as possible. When using a font with multiple height versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.AutoWidthChar(c, width, state, always=False, char_class=<class 'matplotlib.mathtext.Char'>)
   Bases: matplotlib.mathtext.Hlist

   AutoWidthChar will create a character as close to the given width as possible. When using a font with multiple width versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.BakomaFonts(*args, **kwargs)
   Bases: matplotlib.mathtext.TruetypeFonts

   Use the Bakoma TrueType fonts for rendering.

   Symbols are strewn about a number of font files, each of which has its own proprietary 8-bit encoding.
alias = u'\l']'

get_sized_alternatives_for_symbol(fontname, sym)

target = u']'

class matplotlib.mathtext.Box(width, height, depth)
   Bases: matplotlib.mathtext.Node
   Represents any node with a physical location.
   grow()
   render(x1, y1, x2, y2)
   shrink()

class matplotlib.mathtext.Char(c, state)
   Bases: matplotlib.mathtext.Node
   Represents a single character. Unlike TeX, the font information and metrics are stored with each Char to make it easier to lookup the font metrics when needed. Note that TeX boxes have a width, height, and depth, unlike Type1 and Truetype which use a full bounding box and an advance in the x-direction. The metrics must be converted to the TeX way, and the advance (if different from width) must be converted into a Kern node when the Char is added to its parent Hlist.
   get_kerning(next)
      Return the amount of kerning between this and the given character. Called when characters are strung together into Hlist to create Kern nodes.
   grow()
   is_slanted()
   render(x, y)
      Render the character to the canvas
   shrink()

matplotlib.mathtext.Error(msg)
   Helper class to raise parser errors.

class matplotlib.mathtext.Fill
   Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Fill
   Bases: matplotlib.mathtext.Glue
class matplotlib.mathtext.Fill

    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Fonts(
    default_font_prop, mathtext_backend
)

    Bases: object

    An abstract base class for a system of fonts to use for mathtext.
    The class must be able to take symbol keys and font file names and return the character metrics. It also delegates to a backend class to do the actual drawing.

    default_font_prop: A FontProperties object to use for the default non-math font, or the base font for Unicode (generic) font rendering.

    mathtext_backend: A subclass of MathTextBackend used to delegate the actual rendering.

def destroy()

    Fix any cyclical references before the object is about to be destroyed.

def get_kern(
    font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi
)

    Get the kerning distance for font between sym1 and sym2.

    fontX: one of the TeX font names:

    tt, it, rm, cal, sf, bf or default/regular (non-math)

    fontclassX: TODO

    symX: a symbol in raw TeX form. e.g., ‘1’, ‘x’ or ‘sigma’

    fontsizeX: the fontsize in points

    dpi: the current dots-per-inch

def get_metrics(
    font, font_class, sym, fontsize, dpi
)

    font: one of the TeX font names:

    tt, it, rm, cal, sf, bf or default/regular (non-math)

    font_class: TODO

    sym: a symbol in raw TeX form. e.g., ‘1’, ‘x’ or ‘sigma’

    fontsize: font size in points

    dpi: current dots-per-inch

    Returns an object with the following attributes:

    - advance: The advance distance (in points) of the glyph.
    - height: The height of the glyph in points.
    - width: The width of the glyph in points.
    - xmin, xmax, ymin, ymax - the ink rectangle of the glyph
    - iceberg - the distance from the baseline to the top of the glyph. This corresponds to TeX’s definition of “height”.

get_results(box)
Get the data needed by the backend to render the math expression. The return value is backend-specific.

get_sized_alternatives_for_symbol(fontname, sym)
Override if your font provides multiple sizes of the same symbol. Should return a list of symbols matching sym in various sizes. The expression renderer will select the most appropriate size for a given situation from this list.

get_underline_thickness(font, fontsize, dpi)
Get the line thickness that matches the given font. Used as a base unit for drawing lines such as in a fraction or radical.

get_used_characters()
Get the set of characters that were used in the math expression. Used by backends that need to subset fonts so they know which glyphs to include.

get_xheight(font, fontsize, dpi)
Get the xheight for the given font and fontsize.

render_glyph(ox, oy, facename, font_class, sym, fontsize, dpi)
Draw a glyph at
• ox, oy: position
• facename: One of the TeX face names
• font_class:
• sym: TeX symbol name or single character
• fontsize: fontsize in points
• dpi: The dpi to draw at.

render_rect_filled(x1, y1, x2, y2)
Draw a filled rectangle from (x1, y1) to (x2, y2).

set_canvas_size(w, h, d)
Set the size of the buffer used to render the math expression. Only really necessary for the bitmap backends.

class matplotlib.mathtext.Glue(glue_type, copy=False)
Bases: matplotlib.mathtext.Node
Most of the information in this object is stored in the underlying GlueSpec class, which is shared between multiple glue objects. (This is a memory optimization which probably doesn’t matter anymore, but it’s easier to stick to what TeX does.)
grow()

shrink()

class matplotlib.mathtext.GlueSpec(width=0.0, stretch=0.0, stretch_order=0, shrink=0.0, shrink_order=0)
Bases: object
See Glue.
copy()

classmethod factory(glue_type)

class matplotlib.mathtext.HCentered(elements)
    Bases: matplotlib.mathtext.Hlist
    
A convenience class to create an Hlist whose contents are centered within its enclosing box.

class matplotlib.mathtext.Hbox(width)
    Bases: matplotlib.mathtext.Box
    
A box with only width (zero height and depth).

class matplotlib.mathtext.Hlist(elements, w=0.0, m=u'additional', do_kern=True)
    Bases: matplotlib.mathtext.List
    
A horizontal list of boxes.

hpack(w=0.0, m=u'additional')
    
The main duty of hpack() is to compute the dimensions of the resulting boxes, and to adjust
the glue if one of those dimensions is pre-specified. The computed sizes normally enclose all
of the material inside the new box; but some items may stick out if negative glue is used, if the
box is overfull, or if a \vbox includes other boxes that have been shifted left.

• w: specifies a width
• m: is either 'exactly' or 'additional'.

Thus, hpack(w, 'exactly') produces a box whose width is exactly w, while hpack(w, 'additional') yields a box whose width is the natural width plus w. The default values
produce a box with the natural width.

kern()
    
Insert Kern nodes between Char nodes to set kerning. The Char nodes themselves determine
the amount of kerning they need (in get_kerning()), and this function just creates the linked
list in the correct way.

class matplotlib.mathtext.Hrule(state, thickness=None)
    Bases: matplotlib.mathtext.Rule
    
Convenience class to create a horizontal rule.

class matplotlib.mathtext.Kern(width)
    Bases: matplotlib.mathtext.Node
    
A Kern node has a width field to specify a (normally negative) amount of spacing. This spacing
correction appears in horizontal lists between letters like A and V when the font designer said that it
looks better to move them closer together or further apart. A kern node can also appear in a vertical
list, when its width denotes additional spacing in the vertical direction.

depth = 0

grow()
height = 0

shrink()

class matplotlib.mathtext.List(elements)
    Bases: matplotlib.mathtext.Box
    A list of nodes (either horizontal or vertical).

grow()

shrink()

class matplotlib.mathtext.MathTextParser(output)
    Bases: object
    Create a MathTextParser for the given backend output.

get_depth(texstr, dpi=120, fontsize=14)
    Returns the offset of the baseline from the bottom of the image in pixels.
    texstr A valid mathtext string, e.g., r'IQ: $\sigma_i = 15$'
    dpi The dots-per-inch to render the text
    fontsize The font size in points

parse(s, dpi=72, prop=None)
    Parse the given math expression s at the given dpi. If prop is provided, it is a FontProperties
    object specifying the “default” font to use in the math expression, used for all non-math text.
    The results are cached, so multiple calls to parse() with the same expression should be fast.

to_mask(texstr, dpi=120, fontsize=14)
    texstr A valid mathtext string, e.g., r'IQ: $\sigma_i = 15$'
    dpi The dots-per-inch to render the text
    fontsize The font size in points
    Returns a tuple (array, depth)
    • array is an NxM uint8 alpha ubyte mask array of rasterized tex.
    • depth is the offset of the baseline from the bottom of the image in pixels.

to_png(filename, texstr, color='black', dpi=120, fontsize=14)
    Writes a tex expression to a PNG file.

    Returns the offset of the baseline from the bottom of the image in pixels.
    filename A writable filename or fileobject
    texstr A valid mathtext string, e.g., r'IQ: $\sigma_i = 15$'
    color A valid matplotlib color argument
    dpi The dots-per-inch to render the text
    fontsize The font size in points
    Returns the offset of the baseline from the bottom of the image in pixels.
to_rgba(texstr, color='black', dpi=120, fontsize=14)

- **texstr**: A valid mathtext string, e.g., r'IQ: $\sigma_i = 15$'
- **color**: Any matplotlib color argument
- **dpi**: The dots-per-inch to render the text
- **fontsize**: The font size in points

Returns a tuple (array, depth)
- **array**: an N×M uint8 alpha ubyte mask array of rasterized tex.
- **depth**: is the offset of the baseline from the bottom of the image in pixels.

**exception** matplotlib.mathtext.MathTextWarning

Bases: exceptions.Warning

class matplotlib.mathtext.MathtextBackend

Bases: object

The base class for the mathtext backend-specific code. The purpose of **MathtextBackend** subclasses is to interface between mathtext and a specific matplotlib graphics backend.

Subclasses need to override the following:
- **render_glyph()**
- **render_rect_filled()**
- **get_results()**

And optionally, if you need to use a Freetype hinting style:
- **get_hinting_type()**

**get_hinting_type()**

Get the Freetype hinting type to use with this particular backend.

**get_results**(box)

Return a backend-specific tuple to return to the backend after all processing is done.

**render_glyph**(ox, oy, info)

Draw a glyph described by info to the reference point (ox, oy).

**render_rect_filled**(x1, y1, x2, y2)

draw a filled black rectangle from (x1, y1) to (x2, y2).

**set_canvas_size**(w, h, d)

Dimension the drawing canvas.

class matplotlib.mathtext.MathtextBackendAgg

Bases: matplotlib.mathtext.MathtextBackend

Render glyphs and rectangles to an FTImage buffer, which is later transferred to the Agg image by the Agg backend.

**get_hinting_type()**

**get_results**(box, used_characters)

**render_glyph**(ox, oy, info)
render_rect_filled($x_1, y_1, x_2, y_2$)

set_canvas_size($w, h, d$)

class matplotlib.mathtext.MathtextBackendBitmap
    Bases: matplotlib.mathtext.MathtextBackendAgg
    get_results($box, used_characters$)

class matplotlib.mathtext.MathtextBackendCairo
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the Cairo backend.
    get_results($box, used_characters$)
    render_glyph($ox, oy, info$)
    render_rect_filled($x_1, y_1, x_2, y_2$)

class matplotlib.mathtext.MathtextBackendPath
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the text path machinery.
    get_results($box, used_characters$)
    render_glyph($ox, oy, info$)
    render_rect_filled($x_1, y_1, x_2, y_2$)

class matplotlib.mathtext.MathtextBackendPdf
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the PDF backend.
    get_results($box, used_characters$)
    render_glyph($ox, oy, info$)
    render_rect_filled($x_1, y_1, x_2, y_2$)

class matplotlib.mathtext.MathtextBackendPs
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the PostScript backend.
get_results(box, used_characters)

render_glyph(ox, oy, info)

render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathTextBackendSvg
    Bases: matplotlib.mathtext.MathTextBackend
    Store information to write a mathtext rendering to the SVG backend.
    get_results(box, used_characters)

    renderGlyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.NegFil
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.NegFill
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.NegFilll
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Node
    Bases: object
    A node in the TeX box model
    get_kerning(next)

grow()
    Grows one level larger. There is no limit to how big something can get.

render(x, y)

shrink()
    Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

class matplotlib.mathtext.Parser
    Bases: object
    This is the pyparsing-based parser for math expressions. It actually parses full strings containing math expressions, in that raw text may also appear outside of pairs of $.

    The grammar is based directly on that in TeX, though it cuts a few corners.
class `State`(font_output, font, font_class, fontsize, dpi)

Bases: `object`

Stores the state of the parser.

States are pushed and popped from a stack as necessary, and the “current” state is always at the top of the stack.

`copy()`

`font`

Parser. `accent`(s, loc, toks)

Parser. `auto_delim`(s, loc, toks)

Parser. `binom`(s, loc, toks)

Parser. `c_over_c`(s, loc, toks)

Parser. `customspace`(s, loc, toks)

Parser. `end_group`(s, loc, toks)

Parser. `font`(s, loc, toks)

Parser. `frac`(s, loc, toks)

Parser. `function`(s, loc, toks)

Parser. `genfrac`(s, loc, toks)

Parser. `get_state`()  
Get the current `State` of the parser.

Parser. `group`(s, loc, toks)

Parser. `is_dropsub`(nucleus)

Parser. `is_overunder`(nucleus)
Parser.is_slanted(nucleus)

Parser.main(s, loc, toks)

Parser.math(s, loc, toks)

Parser.math_string(s, loc, toks)

Parser.non_math(s, loc, toks)

Parser.operatorname(s, loc, toks)

Parser.overline(s, loc, toks)

Parser.parse(s, fonts_object, fontsize, dpi)
    Parse expression s using the given fonts_object for output, at the given fontsize and dpi.
    Returns the parse tree of Node instances.

Parser.pop_state()
    Pop a State off of the stack.

Parser.push_state()
    Push a new State onto the stack which is just a copy of the current state.

Parser.required_group(s, loc, toks)

Parser.simple_group(s, loc, toks)

Parser.space(s, loc, toks)

Parser.sqrt(s, loc, toks)

Parser.stackrel(s, loc, toks)

Parser.start_group(s, loc, toks)

Parser.subsuper(s, loc, toks)

Parser.symbol(s, loc, toks)
Parser.unknown_symbol(s, loc, toks)

class matplotlib.mathtext.Rule(width, height, depth, state)
   Bases: matplotlib.mathtext.Box

   A Rule node stands for a solid black rectangle; it has width, depth, and height fields just as in an Hlist. However, if any of these dimensions is inf, the actual value will be determined by running the rule up to the boundary of the innermost enclosing box. This is called a “running dimension.” The width is never running in an Hlist; the height and depth are never running in a Vlist.

   render(x, y, w, h)

class matplotlib.mathtext.Ship
   Bases: object

   Once the boxes have been set up, this sends them to output. Since boxes can be inside of boxes inside of boxes, the main work of Ship is done by two mutually recursive routines, hlist_out() and vlist_out(), which traverse the Hlist nodes and Vlist nodes inside of horizontal and vertical boxes. The global variables used in TeX to store state as it processes have become member variables here.

   static clamp(value)

   hlist_out(box)

   vlist_out(box)

class matplotlib.mathtext.SsGlue
   Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.StandardPsFonts(default_font_prop)
   Bases: matplotlib.mathtext.Fonts

   Use the standard postscript fonts for rendering to backend_ps

   Unlike the other font classes, BakomaFont and UnicodeFont, this one requires the Ps backend.

   basepath = u'/home/tcaswell/virtualenvs/mpl_doc_build/lib/python2.7/site-packages/matplotlib-1.4.3-py2.7-linux-x86_64.egg/matplotlib/mpl-data/fonts/afm'

   fontmap = {u'bf': u'pncb8a', u'tt': u'pcrr8a', u'it': u'pncri8a', None: u'psyr', u'cal': u'pzcmi8a', u'rm': u'pncr8a'}

   get_kern(font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi)

   get_underline_thickness(font, fontsize, dpi)

   get_xheight(font, fontsize, dpi)
class matplotlib.mathtext.StixFonts(*args, **kwargs)
    Bases: matplotlib.mathtext.UnicodeFonts

    A font handling class for the STIX fonts.

    In addition to what UnicodeFonts provides, this class:
    • supports “virtual fonts” which are complete alpha numeric character sets with different font styles at special Unicode code points, such as “Blackboard”.
    • handles sized alternative characters for the STIXSizeX fonts.

    cm_fallback = False

    get_sized_alternatives_for_symbol(fontname, sym)

    use_cmex = False

class matplotlib.mathtext.StixSansFonts(*args, **kwargs)
    Bases: matplotlib.mathtext.StixFonts

    A font handling class for the STIX fonts (that uses sans-serif characters by default).

class matplotlib.mathtext.SubSuperCluster
    Bases: matplotlib.mathtext.Hlist

    SubSuperCluster is a sort of hack to get around that fact that this code do a two-pass parse like TeX. This lets us store enough information in the hlist itself, namely the nucleus, sub- and super-script, such that if another script follows that needs to be attached, it can be reconfigured on the fly.

class matplotlib.mathtext.TruetypeFonts(default_font_prop, mathtext_backend)
    Bases: matplotlib.mathtext.Fonts

    A generic base class for all font setups that use Truetype fonts (through FT2Font).

class CachedFont(font)

    TruetypeFonts.destroy()

    TruetypeFonts.get_kern(font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi)

    TruetypeFonts.get_underline_thickness(font, fontsize, dpi)

    TruetypeFonts.get_xheight(font, fontsize, dpi)

class matplotlib.mathtext.UnicodeFonts(*args, **kwargs)
    Bases: matplotlib.mathtext.TruetypeFonts

    An abstract base class for handling Unicode fonts.
While some reasonably complete Unicode fonts (such as DejaVu) may work in some situations, the only Unicode font I’m aware of with a complete set of math symbols is STIX.

This class will “fallback” on the Bakoma fonts when a required symbol can not be found in the font.

\texttt{get\_sized\_alternatives\_for\_symbol}(\texttt{fontname}, \texttt{sym})

\texttt{use\_cmex} = True

class \texttt{matplotlib.mathtext.VCentered}(\texttt{elements})
  Bases: \texttt{matplotlib.mathtext.Hlist}

A convenience class to create a Vlist whose contents are centered within its enclosing box.

class \texttt{matplotlib.mathtext.Vbox}(\texttt{height}, \texttt{depth})
  Bases: \texttt{matplotlib.mathtext.Box}

A box with only height (zero width).

class \texttt{matplotlib.mathtext.Vlist}(\texttt{elements}, \texttt{h=0.0}, \texttt{m=u’additional’})
  Bases: \texttt{matplotlib.mathtext.List}

A vertical list of boxes.

\texttt{vpack}(\texttt{h=0.0}, \texttt{m=u’additional’}, \texttt{l=inf})

The main duty of \texttt{vpack()} is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified.

- \texttt{h}: specifies a height
- \texttt{m}: is either ‘exactly’ or ‘additional’.
- \texttt{l}: a maximum height

Thus, \texttt{vpack(h, ‘exactly’)} produces a box whose height is exactly \texttt{h}, while \texttt{vpack(h, ‘additional’)} yields a box whose height is the natural height plus \texttt{h}. The default values produce a box with the natural width.

class \texttt{matplotlib.mathtext.Vrule}(\texttt{state})
  Bases: \texttt{matplotlib.mathtext.Rule}

Convenience class to create a vertical rule.

\texttt{matplotlib.mathtext.get\_unicode\_index}(\texttt{symbol}) \rightarrow \text{integer}

Return the integer index (from the Unicode table) of symbol. \texttt{symbol} can be a single unicode character, a TeX command (i.e. \texttt{r’pi’}), or a Type1 symbol name (i.e. ‘phi’).

\texttt{matplotlib.mathtext.math\_to\_image}(\texttt{s}, \texttt{filename\_or\_obj}, \texttt{prop=None}, \texttt{dpi=None}, \texttt{format=None})

Given a math expression, renders it in a closely-clipped bounding box to an image file. \texttt{s} A math expression. The math portion should be enclosed in dollar signs.

- \texttt{filename\_or\_obj} A filepath or writable file-like object to write the image data to.
- \texttt{prop} If provided, a FontProperties() object describing the size and style of the text.
- \texttt{dpi} Override the output dpi, otherwise use the default associated with the output format.
- \texttt{format} The output format, e.g., ‘svg’, ‘pdf’, ‘ps’ or ‘png’. If not provided, will be deduced from the filename.
matplotlib.mathtext.unichr_safe(index)

Return the Unicode character corresponding to the index, or the replacement character if this is a narrow build of Python and the requested character is outside the BMP.
58.1 matplotlib.mlab

Numerical python functions written for compatibility with MATLAB commands with the same names.

58.1.1 MATLAB compatible functions

- **cohere()**  Coherence (normalized cross spectral density)
- **csd()**  Cross spectral density using Welch’s average periodogram
- **detrend()**  Remove the mean or best fit line from an array
- **find()**  Return the indices where some condition is true; numpy.nonzero is similar but more general.
- **griddata()**  Interpolate irregularly distributed data to a regular grid.
- **prctile()**  Find the percentiles of a sequence
- **prepca()**  Principal Component Analysis
- **psd()**  Power spectral density using Welch’s average periodogram
- **rk4()**  A 4th order Runge-Kutta integrator for 1D or ND systems
- **specgram()**  Spectrogram (spectrum over segments of time)

58.1.2 Miscellaneous functions

Functions that don’t exist in MATLAB, but are useful anyway:

- **cohere_pairs()**  Coherence over all pairs. This is not a MATLAB function, but we compute coherence a lot in my lab, and we compute it for a lot of pairs. This function is optimized to do this efficiently by caching the direct FFTs.
- **rk4()**  A 4th order Runge-Kutta ODE integrator in case you ever find yourself stranded without scipy (and the far superior scipy.integrate tools)
- **contiguous_regions()**  Return the indices of the regions spanned by some logical mask
- **cross_from_below()**  Return the indices where a 1D array crosses a threshold from below
**cross_from_above()**  Return the indices where a 1D array crosses a threshold from above

**complex_spectrum()**  Return the complex-valued frequency spectrum of a signal

**magnitude_spectrum()**  Return the magnitude of the frequency spectrum of a signal

**angle_spectrum()**  Return the angle (wrapped phase) of the frequency spectrum of a signal

**phase_spectrum()**  Return the phase (unwrapped angle) of the frequency spectrum of a signal

**detrend_mean()**  Remove the mean from a line.

**demean()**  Remove the mean from a line. This function is the same as as **detrend_mean()** except for the default axis.

**detrend_linear()**  Remove the best fit line from a line.

**detrend_none()**  Return the original line.

**stride_windows()**  Get all windows in an array in a memory-efficient manner

**stride_repeat()**  Repeat an array in a memory-efficient manner

**apply_window()**  Apply a window along a given axis

### 58.1.3 record array helper functions

A collection of helper methods for numpy record arrays

See misc Examples

**rec2txt()**  Pretty print a record array

**rec2csv()**  Store record array in CSV file

**csv2rec()**  Import record array from CSV file with type inspection

**rec_append_fields()**  Adds field(s)/array(s) to record array

**rec_drop_fields()**  Drop fields from record array

**rec_join()**  Join two record arrays on sequence of fields

**recs_join()**  A simple join of multiple recarrays using a single column as a key

**rec_groupby()**  Summarize data by groups (similar to SQL GROUP BY)

**rec_summarize()**  Helper code to filter rec array fields into new fields

For the rec viewer functions(e rec2csv), there are a bunch of Format objects you can pass into the functions that will do things like color negative values red, set percent formatting and scaling, etc.

Example usage:

```python
r = csv2rec('somefile.csv', checkrows=0)

formatd = dict(
    weight = FormatFloat(2),
    change = FormatPercent(2),
```
cost = FormatThousands(2),

rec2excel(r, 'test.xls', formatd=formatd)
rec2csv(r, 'test.csv', formatd=formatd)
scroll = rec2gtk(r, formatd=formatd)

win = gtk.Window()
win.set_size_request(600,800)
win.add(scroll)
win.show_all()
gtk.main()

58.1.4 Deprecated functions

The following are deprecated; please import directly from numpy (with care–function signatures may differ):

load()  Load ASCII file - use numpy.loadtxt
save()  Save ASCII file - use numpy.savetxt

class matplotlib.mlab.FIFOBuffer(*args, **kwargs)
A FIFO queue to hold incoming x, y data in a rotating buffer using numpy arrays under the hood. It is assumed that you will call asarrays much less frequently than you add data to the queue – otherwise another data structure will be faster.

This can be used to support plots where data is added from a real time feed and the plot object wants to grab data from the buffer and plot it to screen less frequently than the incoming.

If you set the dataLim attr to BBox (e.g., matplotlib.Axes.dataLim), the dataLim will be updated as new data come in.

TODO: add a grow method that will extend nmax

**Note:** mlab seems like the wrong place for this class.

Deprecated since version 1.3: The FIFOBuffer class was deprecated in version 1.3.

Buffer up to nmax points.

add(x, y)
Add scalar x and y to the queue.

asarrays()
Return x and y as arrays; their length will be the len of data added or nmax.

last()
Get the last x, y or None. None if no data set.

register(func, N)
Call func every time N events are passed; func signature is func(fifo).
update_datalim_to_current()

Update the datalim in the current data in the fifo.

class matplotlib.mlab.FormatBool
    Bases: matplotlib.mlab.FormatObj

    fromstr(s)

    toval(x)

class matplotlib.mlab.FormatDate(fmt)
    Bases: matplotlib.mlab.FormatObj

    fromstr(x)

    toval(x)

class matplotlib.mlab.FormatDatetime(fmt=u'%Y-%m-%d %H:%M:%S')
    Bases: matplotlib.mlab.FormatDate

    fromstr(x)

class matplotlib.mlab.FormatFloat(precision=4, scale=1.0)
    Bases: matplotlib.mlab.FormatFormatStr

    fromstr(s)

    toval(x)

class matplotlib.mlab.FormatFormatStr(fmt)
    Bases: matplotlib.mlab.FormatObj

    tostr(x)

class matplotlib.mlab.FormatInt
    Bases: matplotlib.mlab.FormatObj

    fromstr(s)

    tostr(x)

    toval(x)

class matplotlib.mlab.FormatMillions(precision=4)
    Bases: matplotlib.mlab.FormatFloat
class matplotlib.mlab.FormatObj

    fromstr(s)

    tostr(x)

    toval(x)

class matplotlib.mlab.FormatPercent(precision=4)
    Bases: matplotlib.mlab.FormatFloat

class matplotlib.mlab.FormatString
    Bases: matplotlib.mlab.FormatObj

    tostr(x)

class matplotlib.mlab.FormatThousands(precision=4)
    Bases: matplotlib.mlab.FormatFloat

class matplotlib.mlab.GaussianKDE(dataset, bw_method=None)
    Bases: object

    Representation of a kernel-density estimate using Gaussian kernels.

    Call signature: kde = GaussianKDE(dataset, bw_method='silverman')

    Parameters dataset : array_like
        Datapoints to estimate from. In case of univariate data this is a 1-D
        array, otherwise a 2-D array with shape (# of dims, # of data).

    bw_method : str, scalar or callable, optional
        The method used to calculate the estimator bandwidth. This can be
        ’scott’, ’silverman’, a scalar constant or a callable. If a scalar, this
        will be used directly as kde.factor. If a callable, it should take a
        GaussianKDE instance as only parameter and return a scalar. If None
        (default), ’scott’ is used.

    Attributes

    dataset : (ndarray) The dataset with which gaussian_kde was initialized.
    dim : (int) Number of dimensions.
    num_dp : (int) Number of datapoints.
    factor : (float) The bandwidth factor, obtained from kde.covariance_factor, with which
        the covariance matrix is multiplied.
    covariance : (ndarray) The covariance matrix of dataset, scaled by the calculated bandwidth
        (kde.factor).
    inv_cov : (ndarray) The inverse of covariance.
Methods

<table>
<thead>
<tr>
<th>kde.evaluate(points)</th>
<th>(ndarray) Evaluate the estimated pdf on a provided set of points.</th>
</tr>
</thead>
<tbody>
<tr>
<td>kde(points)</td>
<td>(ndarray) Same as kde.evaluate(points)</td>
</tr>
</tbody>
</table>

**covariance_factor()**

**evaluate(points)**

Evaluate the estimated pdf on a set of points.

Parameters:
- **points** : (# of dimensions, # of points)-array
  
  Optionally, a (# of dimensions,) vector can be passed in and treated as a single point.

Returns:
- **values** : (# of points,)-array
  
  The values at each point.

Raises **ValueError** : if the dimensionality of the input points is different than the dimensionality of the KDE.

**scotts_factor()**

**silverman_factor()**

class matplotlib.mlab.PCA(a, standardize=True)

compute the SVD of a and store data for PCA. Use project to project the data onto a reduced set of dimensions

Inputs:
- **a** : a numobservations x numdims array
  
  **standardize** : True if input data are to be standardized. If False, only centering will be carried out.

Attrs:
- **a** : a centered unit sigma version of input a
  
  **numrows, numcols** : the dimensions of a

- **mu** : a numdims array of means of a. This is the vector that points to the origin of PCA space.

- **sigma** : a numdims array of standard deviation of a

- **fracs** : the proportion of variance of each of the principal components

- **s** : the actual eigenvalues of the decomposition

- **Wt** : the weight vector for projecting a numdims point or array into PCA space

- **Y** : a projected into PCA space

The factor loadings are in the Wt factor, i.e., the factor loadings for the 1st principal component are given by Wt[0]. This row is also the 1st eigenvector.

**center(x)**

center and optionally standardize the data using the mean and sigma from training set a
project\(x, \text{minfrac}=0.0\)

project \(x\) onto the principle axes, dropping any axes where fraction of variance < minfrac

```
matplotlib.mlab.amap(function, sequence[, sequence, ...]) -> array.
```

Works like `map()`, but it returns an array. This is just a convenient shorthand for `numpy.array(map(...))`.

```
matplotlib.mlab.angle_spectrum(x, Fs=None, window=None, pad_to=None, sides=None)
```

Compute the angle of the frequency spectrum (wrapped phase spectrum) of \(x\). Data is padded to a length of `pad_to` and the windowing function `window` is applied to the signal.

- **\(x\): 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

- **Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
- **window**: callable or ndarray A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
- **sides**: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.
- **pad_to**: integer The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \(n\) parameter in the call to `fft()`. The default is None, which sets `pad_to` equal to the length of the input signal (i.e. no padding).

Returns the tuple \((\text{spectrum}, \text{freqs})\):

- **\(\text{spectrum\): 1-D array** The values for the angle spectrum in radians (real valued)
- **\(\text{freqs\): 1-D array** The frequencies corresponding to the elements in `\spectrum`

See also:

- `complex_spectrum()` This function returns the angle value of `complex_spectrum()`.
- `magnitude_spectrum()` `angle_spectrum()` returns the magnitudes of the corresponding frequencies.
- `phase_spectrum()` `phase_spectrum()` returns the unwrapped version of this function.
- `specgram()` `specgram()` can return the angle spectrum of segments within the signal.

```
matplotlib.mlab.apply_window(x, window, axis=0, return_window=None)
```

Apply the given window to the given 1D or 2D array along the given axis.

Call signature:

```
apply_window(x, window, axis=0, return_window=False)
```

*x*: 1D or 2D array or sequence

Array or sequence containing the data.
*winodw*: function or array.
   Either a function to generate a window or an array with length
   *x*.shape[*axis*]

*axis*: integer
   The axis over which to do the repetition.
   Must be 0 or 1. The default is 0

*return_window*: bool
   If true, also return the 1D values of the window that was applied

matplotlib.mlab.base_repr(number, base=2, padding=0)
   Return the representation of a number in any given base.

matplotlib.mlab.binary_repr(number, max_length=1025)
   Return the binary representation of the input number as a string.
   This is more efficient than using base_repr() with base 2.
   Increase the value of max_length for very large numbers. Note that on 32-bit machines, 2**1023 is
   the largest integer power of 2 which can be converted to a Python float.

matplotlib.mlab.bivariate_normal(X, Y, sigmax=1.0, sigmay=1.0, mux=0.0, muy=0.0, sigmaxy=0.0)
   Bivariate Gaussian distribution for equal shape X, Y.
   See bivariate normal at mathworld.

matplotlib.mlab.center_matrix(M, dim=0)
   Return the matrix M with each row having zero mean and unit std.
   If dim = 1 operate on columns instead of rows. (dim is opposite to the numpy axis kwarg.)

matplotlib.mlab.cohere(x, y, NFFT=256, Fs=2, detrend=<function detrend_none at 0x7f8bf872f320>,
                             window=<function window_hanning at 0x7f8bf872f050>,
                             noverlap=0, pad_to=None, sides='default',
                             scale_by_freq=None)
   The coherence between x and y. Coherence is the normalized cross spectral density:

   $C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$  \hspace{1cm} (58.1)

   x, y Array or sequence containing the data
   Keyword arguments:
   Fs: scalar The sampling frequency (samples per time unit). It is used to calculate the
   Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
   window: callable or ndarray A function or a vector of length NFFT. To
   create window vectors see window_hanning(), window_none(),
   numpy.blackman(), numpy.hamming(), numpy.bartlett(),
   scipy.signal(), scipy.signal.get_window(), etc. The default is
   window_hanning(). If a function is passed as the argument, it must take a
   data segment as an argument and return the windowed version of the segment.
sides: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

pad_to: integer The number of points to which the data segment is padded when performing the FFT. This can be different from NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to NFFT

NFFT: integer The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

detrend: [‘default’ | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’] or callable
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

scale_by_freq: boolean
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap: integer The number of points of overlap between blocks. The default value is 0 (no overlap).

The return value is the tuple (Cxy, f), where f are the frequencies of the coherence vector. For cohere, scaling the individual densities by the sampling frequency has no effect, since the factors cancel out.

See also:

psd() and csd() For information about the methods used to compute P_{xy}, P_{xx} and P_{yy}.

```python
matplotlib.mlab.coherence_pairs(X, i, j, NFFT=256, Fs=2, detrend=<function detrend_none at 0x7f8bf872f320>, window=<function window_hanning at 0x7f8bf872f050>, noverlap=0, preferSpeedOverMemory=True, progressCallback=<function donothing_callback at 0x7f8bf872faa0>, returnPxx=False)
```

Call signature:

```python
Cxy, Phase, freqs = coherence_pairs(X, i, j, ...) 
```

Compute the coherence and phase for all pairs i, j, in X.

X is a numSamples * numCols array

i, j is a list of tuples. Each tuple is a pair of indexes into the columns of X for which you want to compute coherence. For example, if X has 64 columns, and you want to compute all nonredundant pairs, define i, j as:
ij = []
for i in range(64):
    for j in range(i+1,64):
        ij.append((i,j))

preferSpeedOverMemory is an optional bool. Defaults to true. If False, limits the caching by only making one, rather than two, complex cache arrays. This is useful if memory becomes critical. Even when preferSpeedOverMemory is False, cohere_pairs() will still give significant performance gains over calling cohere() for each pair, and will use substantially less memory than if preferSpeedOverMemory is True. In my tests with a 43000,64 array over all nonredundant pairs, preferSpeedOverMemory = True delivered a 33% performance boost on a 1.7GHZ Athlon with 512MB RAM compared with preferSpeedOverMemory = False. But both solutions were more than 10x faster than naively crunching all possible pairs through cohere().

Returns:

(Cxy, Phase, freqs)

where:

• Cxy: dictionary of (i, j) tuples -> coherence vector for that pair. i.e., Cxy[(i,j)] = cohere(X[:,i], X[:,j]). Number of dictionary keys is len(ij).
• Phase: dictionary of phases of the cross spectral density at each frequency for each pair. Keys are (i, j).
• freqs: vector of frequencies, equal in length to either the coherence or phase vectors for any (i,j) key.

e.g., to make a coherence Bode plot:

subplot(211)
plot( freqs, Cxy[(12,19)])
subplot(212)
plot( freqs, Phase[(12,19)])

For a large number of pairs, cohere_pairs() can be much more efficient than just calling cohere() for each pair, because it caches most of the intensive computations. If N is the number of pairs, this function is O(N) for most of the heavy lifting, whereas calling cohere for each pair is O(N^2). However, because of the caching, it is also more memory intensive, making 2 additional complex arrays with approximately the same number of elements as X.

See test/cohere_pairs_test.py in the src tree for an example script that shows that this cohere_pairs() and cohere() give the same results for a given pair.

See also:

psd() For information about the methods used to compute P_{xy}, P_{xx} and P_{yy}.

matplotlib.mlab.complex_spectrum(x, Fs=None, window=None, pad_to=None, sides=None)

Compute the complex-valued frequency spectrum of x. Data is padded to a length of pad_to and the windowing function window is applied to the signal.

x: 1-D array or sequence Array or sequence containing the data

Keyword arguments:
**Fs**: scalar  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**window**: callable or ndarray  A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**: ['default' | 'onesided' | 'twosided']  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to**: integer  The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \( n \) parameter in the call to \( \text{fft}() \). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

Returns the tuple \((spectrum, freqs)\):

- **spectrum**: 1-D array  The values for the complex spectrum (complex valued)
- **freqs**: 1-D array  The frequencies corresponding to the elements in spectrum

See also:

- magnitude_spectrum()  magnitude_spectrum() returns the absolute value of this function.
- angle_spectrum()  angle_spectrum() returns the angle of this function.
- phase_spectrum()  phase_spectrum() returns the phase (unwrapped angle) of this function.
- specgram()  specgram() can return the complex spectrum of segments within the signal.

matplotlib.mlab.contiguous_regions(mask)

return a list of (ind0, ind1) such that mask[ind0:ind1].all() is True and we cover all such regions

TODO: this is a pure python implementation which probably has a much faster numpy impl

matplotlib.mlab.cross_from_above(x, threshold)

return the indices into x where x crosses some threshold from below, e.g., the i’s where:

\[ x[i-1] > \text{threshold} \quad \text{and} \quad x[i] \leq \text{threshold} \]

See also:

- cross_from_below() and contiguous_regions()

matplotlib.mlab.cross_from_below(x, threshold)

return the indices into x where x crosses some threshold from below, e.g., the i’s where:

\[ x[i-1] < \text{threshold} \quad \text{and} \quad x[i] \geq \text{threshold} \]

Example code:
import matplotlib.pyplot as plt

t = np.arange(0.0, 2.0, 0.1)
s = np.sin(2*np.pi*t)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(t, s, '-o')
ax.axhline(0.5)
ax.axhline(-0.5)

ind = cross_from_below(s, 0.5)
ax.vlines(t[ind], -1, 1)

ind = cross_from_above(s, -0.5)
ax.vlines(t[ind], -1, 1)

plt.show()

See also:
cross_from_above() and contiguous_regions()

matplotlib.mlab.csd(x, y, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None, pad_to=None, sides=None, scale_by_freq=None)

Compute the cross-spectral density. Call signature:

csd(x, y, NFFT=256, Fs=2, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None)

The cross spectral density $P_{xy}$ by Welch’s average periodogram method. The vectors $x$ and $y$ are divided into $NFFT$ length segments. Each segment is detrended by function $detrend$ and windowed by function $window$. $noverlap$ gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.

If len($x$) < $NFFT$ or len($y$) < $NFFT$, they will be zero padded to $NFFT$.

$x, y$: 1-D arrays or sequences Arrays or sequences containing the data

Keyword arguments:

- $Fs$: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
- $window$: callable or ndarray A function or a vector of length $NFFT$. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(). scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
- $sides$: [ ‘default’ | ‘onesided’ | ‘twosided’ ] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and
both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to**: integer  
The number of points to which the data segment is padded when performing the FFT. This can be different from **NFFT**, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to *fft()*. The default is None, which sets **pad_to** equal to **NFFT**

**NFFT**: integer  
The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use **pad_to** for this instead.

**detrend**: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or callable  
The function applied to each segment before *fft* ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the **detrend** parameter is a vector, in matplotlib it is a function. The *pylab* module defines **detrend_none()**, **detrend_mean()**, and **detrend_linear()**, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call **detrend_mean()**. ‘linear’ calls **detrend_linear()**. ‘none’ calls **detrend_none()**.

**scale_by_freq**: boolean  
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**: integer  
The number of points of overlap between segments. The default value is 0 (no overlap).

Returns the tuple (*Pxy*, *freqs*):

- **Pxy**: 1-D array  
The values for the cross spectrum $P_{xy}$ before scaling (real valued)
- **freqs**: 1-D array  
The frequencies corresponding to the elements in *Pxy*


**See also**:  
**psd()  psd()** is the equivalent to setting y=x.

**matplotlib.mlab.csv2rec**(*fname*, **comments=u'\#', skiprows=0, checkrows=0, delimiter=u',',  
'  converterd=None, names=None, missing=u'', missingd=None,  
  use_mrecords=False, dayfirst=False, yearfirst=False)  
Load data from comma/space/tab delimited file in *fname* into a numpy record array and return the record array.

If **names** is None, a header row is required to automatically assign the recarray names. The headers will be lower cased, spaces will be converted to underscores, and illegal attribute name characters removed. If **names** is not None, it is a sequence of names to use for the column names. In this case, it is assumed there is no header row.

- **fname**: can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in ‘.gz’
- **comments**: the character used to indicate the start of a comment in the file, or None to switch off the removal of comments
• **skiprows**: is the number of rows from the top to skip
• **checkrows**: is the number of rows to check to validate the column data type. When set to zero all rows are validated.
• **converterd**: if not None, is a dictionary mapping column number or munged column name to a converter function.
• **names**: if not None, is a list of header names. In this case, no header will be read from the file
• **missingd**: is a dictionary mapping munged column names to field values which signify that the field does not contain actual data and should be masked, e.g., ‘0000-00-00’ or ‘unused’
• **missing**: a string whose value signals a missing field regardless of the column it appears in
• **use_mrecords**: if True, return an mrecords.fromrecords record array if any of the data are missing
• **dayfirst**: default is False so that MM-DD-YY has precedence over DD-MM-YY. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.
• **yearfirst**: default is False so that MM-DD-YY has precedence over YY-MM-DD. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.

If no rows are found, None is returned – see examples/loadrec.py

**matplotlib.mlab.csvformat_factory**(format)

**matplotlib.mlab.demean**(x, axis=0)

Return x minus its mean along the specified axis.

Call signature:

demean(x, axis=0)

*x*`: array or sequence

Array or sequence containing the data

Can have any dimensionality

*axis*`: integer

The axis along which to take the mean. See numpy.mean for a description of this argument.

See also:

delinear()

denone() delinear() and denone() are other detrend algorithms.
detrend_mean() This function is the same as as detrend_mean() except for the default axis.

**matplotlib.mlab.detrend**(x, key=None, axis=None)

Return x with its trend removed.

Call signature:

detrend(x, key='mean')

*x*`: array or sequence
Array or sequence containing the data.

**key**: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or function

Specifies the detrend algorithm to use. 'default' is 'mean', which is the same as :func:`detrend_mean`. 'constant' is the same. 'linear' is the same as :func:`detrend_linear`. 'none' is the same as :func:`detrend_none`. The default is 'mean'. See the corresponding functions for more details regarding the algorithms. Can also be a function that carries out the detrend operation.

**axis**: integer

The axis along which to do the detrending.

See also:

- :func:`detrend_mean`
- :func:`detrend_linear`
- :func:`detrend_none`

```python
matplotlib.mlab.detrend_linear(y)
```

Return x minus best fit line; 'linear' detrending.

Call signature:

```
detrend_linear(y)
```

**y**: 0-D or 1-D array or sequence

Array or sequence containing the data

**axis**: integer

The axis along which to take the mean. See numpy.mean for a description of this argument.

See also:

- :func:`detrend_mean`
- :func:`detrend_linear`
- :func:`detrend_none`

```python
matplotlib.mlab.detrend_mean(x, axis=None)
```

Return x minus the mean(x).

Call signature:

```
detrend_mean(x, axis=None)
```

**x**: array or sequence

Array or sequence containing the data

**axis**: integer
The axis along which to take the mean. See numpy.mean for a
description of this argument.

See also:

demean() This function is the same as demean() except for the default axis.
detrend_linear()
detrend_none() detrend_linear() and detrend_none() are other detrend algorithms.
detrend() detrend() is a wrapper around all the detrend algorithms.

matplotlib.mlab.detrend_none(x, axis=None)
Return x: no detrending.
Call signature:

detrend_none(x, axis=None)
*x*: any object
An object containing the data

*axis*: integer
This parameter is ignored.
It is included for compatibility with detrend_mean

See also:

denone() This function is the same as denone() except for the default axis, which has no effect.
detrend_mean()
detrend_linear() detrend_mean() and detrend_linear() are other detrend algorithms.
detrend() detrend() is a wrapper around all the detrend algorithms.

matplotlib.mlab.dist(x, y)
Return the distance between two points.

matplotlib.mlab.dist_point_to_segment(p, s0, s1)
Get the distance of a point to a segment.
p, s0, s1 are xy sequences
This algorithm from http://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm#Distance%20to%20Ray%20or%20Segment

matplotlib.mlab.distances_along_curve(X)
Computes the distance between a set of successive points in N dimensions.

Where X is an M x N array or matrix. The distances between successive rows is computed. Distance
is the standard Euclidean distance.

matplotlib.mlab.donthing_callback(*args)

matplotlib.mlab.entropy(y, bins)
Return the entropy of the data in y in units of nat.

\[- \sum p_i \ln(p_i)\]  (58.2)
where $p_i$ is the probability of observing $y$ in the $i$th bin of $bins$. $bins$ can be a number of bins or a range of bins; see numpy.histogram().

Compare $S$ with analytic calculation for a Gaussian:

$$x = \mu + \sigma \cdot \text{randn(200000)}$$

$$S_{\text{analytic}} = 0.5 \cdot \left( 1.0 + \log(2^{\pi*\sigma^2}) \right)$$

matplotlib.mlab.exp_safe($x$)

Compute exponentials which safely underflow to zero.

Slow, but convenient to use. Note that numpy provides proper floating point exception handling with access to the underlying hardware.

matplotlib.mlab.fftsurr($x$, detrend=<function detrend_none at 0x7f8bf872f320>, window=<function window_none at 0x7f8bf872f0c8>)

Compute an FFT phase randomized surrogate of $x$.

matplotlib.mlab.find($condition$)

Return the indices where ravel($condition$) is true.

matplotlib.mlab.frange([start], stop[, step, keywords]) → array of floats

Return a numpy ndarray containing a progression of floats. Similar to numpy.arange(), but defaults to a closed interval.

frange($x0$, $x1$) returns [$x0$, $x0+1$, $x0+2$, ..., $x1$]; start defaults to 0, and the endpoint is included. This behavior is different from that of range() and numpy.arange(). This is deliberate, since frange() will probably be more useful for generating lists of points for function evaluation, and endpoints are often desired in this use. The usual behavior of range() can be obtained by setting the keyword closed = 0, in this case, frange() basically becomes :func:numpy.arange'.

When step is given, it specifies the increment (or decrement). All arguments can be floating point numbers.

frange($x0$, $x1$, $d$) returns [$x0$, $x0+d$, $x0+2d$, ..., $xfin$] where $xfin \leq x1$.

frange() can also be called with the keyword npts. This sets the number of points the list should contain (and overrides the value step might have been given). numpy.arange() doesn't offer this option.

Examples:

```python
>>> frange(3)
array([ 0.,  1.,  2.,  3.])
>>> frange(3,closed=0)
array([ 0.,  1.,  2.])
>>> frange(1,6,2)
array([1, 3, 5]) or 1,3,5,7, depending on floating point vagueries
>>> frange(1,6.5,npts=5)
array([ 1. ,  2.375,  3.75 ,  5.125,  6.5 ])
```

matplotlib.mlab.get_formatd($r$, formatd=None)

build a formatd guaranteed to have a key for every dtype name
matplotlib.mlab.get_sparse_matrix(M, N, frac=0.1)
Return a M x N sparse matrix with frac elements randomly filled.

matplotlib.mlab.get_xyz_where(Z, Cond)
Z and Cond are M x N matrices. Z are data and Cond is a boolean matrix where some condition is satisfied. Return value is (x, y, z) where x and y are the indices into Z and z are the values of Z at those indices. x, y, and z are 1D arrays.

matplotlib.mlab.griddata(x, y, z, xi, yi, interp=u’nn’)
Interpolates from a nonuniformly spaced grid to some other grid.

Fits a surface of the form z = f(x, y) to the data in the (usually) nonuniformly spaced vectors (x, y, z), then interpolates this surface at the points specified by (xi, yi) to produce zi.

Parameters x, y, z : 1d array_like
Coordinates of grid points to interpolate from.
xi, yi : 1d or 2d array_like
Coordinates of grid points to interpolate to.
interp : string key from {‘nn’, ‘linear’}
Interpolation algorithm, either ‘nn’ for natural neighbor, or ‘linear’ for linear interpolation.

Returns 2d float array
Array of values interpolated at (xi, yi) points. Array will be masked is any of (xi, yi) are outside the convex hull of (x, y).

Notes
If interp is ‘nn’ (the default), uses natural neighbor interpolation based on Delaunay triangulation. This option is only available if the mpl_toolkits.natgrid module is installed. This can be downloaded from https://github.com/matplotlib/natgrid. The (xi, yi) grid must be regular and monotonically increasing in this case.

If interp is ‘linear’, linear interpolation is used via matplotlib.tri.LinearTriInterpolator.

Instead of using griddata, more flexible functionality and other interpolation options are available using a matplotlib.tri.Triangulation and a matplotlib.tri.TriInterpolator.

matplotlib.mlab.identity(n, rank=2, dtype=u’l’, typecode=None)
Returns the identity matrix of shape (n, n, ..., n) (rank r).

For ranks higher than 2, this object is simply a multi-index Kronecker delta:

    / 1  if i0=i1=...=iR,
    \ 0  otherwise.

Optionally a dtype (or typecode) may be given (it defaults to ‘l’).

Since rank defaults to 2, this function behaves in the default case (when only n is given) like numpy.identity(n) – but surprisingly, it is much faster.

matplotlib.mlab.inside_poly(points, verts)
points is a sequence of x, y points. verts is a sequence of x, y vertices of a polygon.
Return value is a sequence of indices into points for the points that are inside the polygon.

**matplotlib.mlab.is_closed_polygon(X)**
Tests whether first and last object in a sequence are the same. These are presumably coordinates on a polygonal curve, in which case this function tests if that curve is closed.

**matplotlib.mlab.ispower2(n)**
Returns the log base 2 of n if n is a power of 2, zero otherwise.

Note the potential ambiguity if n == 1: 2**0 == 1, interpret accordingly.

**matplotlib.mlab.isvector(X)**
Like the MATLAB function with the same name, returns True if the supplied numpy array or matrix X looks like a vector, meaning it has a one non-singleton axis (i.e., it can have multiple axes, but all must have length 1, except for one of them).

If you just want to see if the array has 1 axis, use X.ndim == 1.

**matplotlib.mlab.l1norm(a)**
Return the l1 norm of a, flattened out.

Implemented as a separate function (not a call to norm() for speed).

**matplotlib.mlab.l2norm(a)**
Return the l2 norm of a, flattened out.

Implemented as a separate function (not a call to norm() for speed).

**matplotlib.mlab.less_simple_linear_interpolation(x, y, xi, extrap=False)**
This function provides simple (but somewhat less so than cbook.simple_linear_interpolation()) linear interpolation. simple_linear_interpolation() will give a list of point between a start and an end, while this does true linear interpolation at an arbitrary set of points.

This is very inefficient linear interpolation meant to be used only for a small number of points in relatively non-intensive use cases. For real linear interpolation, use scipy.

**matplotlib.mlab.log2(x, ln2=0.6931471805599453)**
Return the log(x) in base 2.

This is a _slow_ function but which is guaranteed to return the correct integer value if the input is an integer exact power of 2.

**matplotlib.mlab.logspace(xmin, xmax, N)**
Return N values logarithmically spaced between xmin and xmax.

Call signature:

logspace(xmin, xmax, N)

**matplotlib.mlab.longest_contiguous_ones(x)**
Return the indices of the longest stretch of contiguous ones in x, assuming x is a vector of zeros and ones. If there are two equally long stretches, pick the first.

**matplotlib.mlab.longest_ones(x)**
alias for longest_contiguous_ones
Compute the magnitude (absolute value) of the frequency spectrum of \( x \). Data is padded to a length of \( \text{pad\_to} \) and the windowing function \( \text{window} \) is applied to the signal.

\[ x: \text{1-D array or sequence} \]
Array or sequence containing the data

Keyword arguments:

- **\( Fs: \) scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **\( \text{window}: \) callable or ndarray** A function or a vector of length \( \text{NFFT} \). To create window vectors see \( \text{window\_hanning()} \), \( \text{window\_none()} \), \( \text{numpy.\_blackman()} \), \( \text{numpy.\_hamming()} \), \( \text{numpy.\_bartlett()} \), \( \text{scipy.\_signal()} \), \( \text{scipy.\_signal.\_get\_window()} \), etc. The default is \( \text{window\_hanning()} \). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **\( \text{sides}: \) \{ ‘default’ | ‘onesided’ | ‘twosided’ \}** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **\( \text{pad\_to}: \) integer** The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \( n \) parameter in the call to \( \text{fft()} \). The default is None, which sets \( \text{pad\_to} \) equal to the length of the input signal (i.e. no padding).

Returns the tuple \( (\text{spectrum}, \text{freqs}) \):

- **\( \text{spectrum}: \) 1-D array** The values for the magnitude spectrum (real valued)
- **\( \text{freqs}: \) 1-D array** The frequencies corresponding to the elements in \( \text{spectrum} \)

See also:

- \( \text{psd()} \): \( \text{psd()} \) returns the power spectral density.
- \( \text{complex\_spectrum()} \): This function returns the absolute value of \( \text{complex\_spectrum()} \).
- \( \text{angle\_spectrum()} \): \( \text{angle\_spectrum()} \) returns the angles of the corresponding frequencies.
- \( \text{phase\_spectrum()} \): \( \text{phase\_spectrum()} \) returns the phase (unwrapped angle) of the corresponding frequencies.
- \( \text{specgram()} \): \( \text{specgram()} \) can return the magnitude spectrum of segments within the signal.

Compute the \( \text{len}(n) \) moving average of \( x \).

\[ \text{norm\_flat}(a, p=2) \]
\[ \text{norm}(a,p=2) \rightarrow \text{1-p norm of a.flat} \]

Return the 1-p norm of \( a \), considered as a flat array. This is NOT a true matrix norm, since arrays of arbitrary rank are always flattened.

\( p \) can be a number or the string ‘Infinity’ to get the L-infinity norm.

Return the normal pdf evaluated at \( x \); \( \text{args} \) provides \( \text{mu, sigma} \)

Return the normal pdf evaluated at \( y \) with \( yerr \) as error bar magnitude.
Offsets an array y by +/- an error and returns a tuple (y - err, y + err).

The error term can be:

- A scalar. In this case, the returned tuple is obvious.
- A vector of the same length as y. The quantities y +/- err are computed component-wise.
- A tuple of length 2. In this case, yerr[0] is the error below y and yerr[1] is error above y. For example:

```python
from pylab import *
x = linspace(0, 2*pi, num=100, endpoint=True)
y = sin(x)
y_minus, y_plus = mlab.offset_line(y, 0.1)
plot(x, y)
fill_between(x, ym, y2=yp)
show()
```

`matplotlib.mlab.path_length(X)`

Computes the distance travelled along a polygonal curve in N dimensions.

Where X is an M x N array or matrix. Returns an array of length M consisting of the distance along the curve at each point (i.e., the rows of X).

`matplotlib.mlab.phase_spectrum(x, Fs=None, window=None, pad_to=None, sides=None)`

Compute the phase of the frequency spectrum (unwrapped angle spectrum) of x. Data is padded to a length of pad_to and the windowing function window is applied to the signal.

- x: 1-D array or sequence Array or sequence containing the data
- Keyword arguments:
  - Fs: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
  - window: callable or ndarray A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
  - sides: ['default' | 'onesided' | ‘twosided’] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.
  - pad_to: integer The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

Returns the tuple (spectrum, freqs):

- spectrum: 1-D array The values for the phase spectrum in radians (real valued)
- freqs: 1-D array The frequencies corresponding to the elements in spectrum

See also:
complex_spectrum() This function returns the angle value of complex_spectrum().
magnitude_spectrum() magnitude_spectrum() returns the magnitudes of the corresponding
frequencies.
angle_spectrum() angle_spectrum() returns the wrapped version of this function.
specgram() specgram() can return the phase spectrum of segments within the signal.

matplotlib.mlab.poly_below(xmin, xs, ys)
Given a sequence of xs and ys, return the vertices of a polygon that has a horizontal base at xmin and
an upper bound at the ys. xmin is a scalar.

Intended for use with matplotlib.axes.Axes.fill(), e.g.:

```
xv, yv = poly_below(0, x, y)
ax.fill(xv, yv)
```

matplotlib.mlab.poly_between(x, ylower, yupper)
Given a sequence of x, ylower and yupper, return the polygon that fills the regions between them.
ylower or yupper can be scalar or iterable. If they are iterable, they must be equal in length to x.

Return value is x, y arrays for use with matplotlib.axes.Axes.fill().

matplotlib.mlab.prctile(x, p=(0.0, 25.0, 50.0, 75.0, 100.0))
Return the percentiles of x. p can either be a sequence of percentile values or a scalar. If p is a
sequence, the ith element of the return sequence is the p*(i)-th percentile of *x. If p is a scalar, the
largest value of x less than or equal to the p percentage point in the sequence is returned.

matplotlib.mlab.prctile_rank(x, p)
Return the rank for each element in x, return the rank 0..len(p). e.g., if p = (25, 50, 75), the return
value will be a len(x) array with values in [0,1,2,3] where 0 indicates the value is less than the 25th
percentile, 1 indicates the value is >= the 25th and < 50th percentile, ... and 3 indicates the value is
above the 75th percentile cutoff.

p is either an array of percentiles in [0..100] or a scalar which indicates how many quantiles of data
you want ranked.

matplotlib.mlab.prepca(P, frac=0)

```
Warning: This function is deprecated – please see class PCA instead
```

Compute the principal components of P. P is a (numVars, numObs) array. frac is the minimum fraction
of variance that a component must contain to be included.

Return value is a tuple of the form (Pcomponents, Trans, fracVar) where:

- Pcomponents : a (numVars, numObs) array
- Trans [the weights matrix, i.e., Pcomponents = Trans *] P
- fracVar [the fraction of the variance accounted for by each] component returned

A similar function of the same name was in the MATLAB R13 Neural Network Toolbox but is not
found in later versions; its successor seems to be called “processpcs”.

matplotlib.mlab.psd(x, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None,

```
pad_to=None, sides=None, scale_by_freq=None)
```

Compute the power spectral density.
Call signature:

```python
psd(x, NFFT=256, Fs=2, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None)
```

The power spectral density $P_{xx}$ by Welch’s average periodogram method. The vector $x$ is divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The $|\text{fft}(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$.

If $\text{len}(x) < NFFT$, it will be zero padded to $NFFT$.

*x*: 1-D array or sequence

Array or sequence containing the data

Keyword arguments:

- **Fs**: scalar
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray
  A function or a vector of length $NFFT$. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: [‘default’ | ‘onesided’ | ‘twosided’ ]
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer
  The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to `fft()`. The default is None, which sets `pad_to` equal to $NFFT$

- **NFFT**: integer
  The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use `pad_to` for this instead.

- **detrend**: [‘default’ | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’] or callable
  The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `detrend` parameter is a vector, in matplotlib it is a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call `detrend_mean()`. ‘linear’ calls `detrend_linear()`. ‘none’ calls `detrend_none()`.

- **scale_by_freq**: boolean
  Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

- **noverlap**: integer
  The number of points of overlap between segments. The default value is 0 (no overlap).
Returns the tuple \((P_{xx}, freqs)\).

**Pxx:** 1-D array The values for the power spectrum \(P_{xx}\) (real valued)

**freqs:** 1-D array The frequencies corresponding to the elements in \(P_{xx}\)

Refs:

See also:
- `specgram()` `specgram()` differs in the default overlap; in not returning the mean of the segment periodograms; and in returning the times of the segments.
- `magnitude_spectrum()` `magnitude_spectrum()` returns the magnitude spectrum.
- `csd()` `csd()` returns the spectral density between two signals.

```python
matplotlib.mlab.quad2cubic(q0x, q0y, q1x, q1y, q2x, q2y)
```
Converting a quadratic Bezier curve to a cubic approximation.

The inputs are the \(x\) and \(y\) coordinates of the three control points of a quadratic curve, and the output is a tuple of \(x\) and \(y\) coordinates of the four control points of the cubic curve.

```python
matplotlib.mlab.rec2csv(r, fname, delimiter=u',' formatd=None, missing=u'', missingd=None, withheader=True)
```
Save the data from numpy recarray \(r\) into a comma-/space-/tab-delimited file. The record array dtype names will be used for column headers.

**fname:** can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in `.gz`

**withheader:** if withheader is False, do not write the attribute names in the first row

For formatd type FormatFloat, we override the precision to store full precision floats in the CSV file

See also:
- `csv2rec()` For information about missing and missingd, which can be used to fill in masked values into your CSV file.

```python
matplotlib.mlab.rec2txt(r, header=None, padding=3, precision=3, fields=None)
```
Returns a textual representation of a record array.

**r:** numpy recarray

**header:** list of column headers

**padding:** space between each column

**precision:** number of decimal places to use for floats. Set to an integer to apply to all floats. Set to a list of integers to apply precision individually. Precision for non-floats is simply ignored.

**fields:** if not None, a list of field names to print. fields can be a list of strings like `[‘field1’, ‘field2’]` or a single comma separated string like ‘field1,field2’

Example:

```python
precision=[0,2,3]
```

Output:
Matplotlib, Release 1.4.3

<table>
<thead>
<tr>
<th>ID</th>
<th>Price</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>12.54</td>
<td>0.234</td>
</tr>
<tr>
<td>XYZ</td>
<td>6.32</td>
<td>-0.076</td>
</tr>
</tbody>
</table>

`matplotlib.mlab.rec_append_fields(rec, names, arrs, dtypes=None)`
Return a new record array with field names populated with data from arrays in `arrs`. If appending a single field, then `names`, `arrs` and `dtypes` do not have to be lists. They can just be the values themselves.

`matplotlib.mlab.rec_drop_fields(rec, names)`
Return a new numpy record array with fields in `names` dropped.

`matplotlib.mlab.rec_groupby(r, groupby, stats)`
r is a numpy record array
groupby is a sequence of record array attribute names that together form the grouping key. e.g., (`'date'`, `'productcode'`)"stats is a sequence of (attr, func, outname) tuples which will call `x = func(attr)` and assign `x` to the record array output with attribute `outname`. For example:

```python
stats = ( ("sales", len, 'numsales'), ("sales", np.mean, 'avgsale') )
```
Return record array has `dtype` names for each attribute name in the the `groupby` argument, with the associated group values, and for each `outname` name in the `stats` argument, with the associated stat summary output.

`matplotlib.mlab.rec_join(key, r1, r2, jointype=u'inner', defaults=None, r1postfix=u'1', r2postfix=u'2')` Join record arrays `r1` and `r2` on `key`; `key` is a tuple of field names – if `key` is a string it is assumed to be a single attribute name. If `r1` and `r2` have equal values on all the keys in the `key` tuple, then their fields will be merged into a new record array containing the intersection of the fields of `r1` and `r2`. `r1` (also `r2`) must not have any duplicate keys.

The `jointype` keyword can be ‘inner’, ‘outer’, ‘leftouter’. To do a rightouter join just reverse `r1` and `r2`.

The `defaults` keyword is a dictionary filled with `{column_name:default_value}` pairs.

The keywords `r1postfix` and `r2postfix` are postfixed to column names (other than keys) that are both in `r1` and `r2`.

`matplotlib.mlab.rec_keep_fields(rec, names)`
Return a new numpy record array with only fields listed in `names`

`matplotlib.mlab.rec_summarize(r, summaryfuncs)`
r is a numpy record array

`summaryfuncs` is a list of (attr, func, outname) tuples which will apply `func` to the the array `r*[attr]` and assign the output to a new attribute name `*outname`. The returned record array is identical to `r`, with extra arrays for each element in `summaryfuncs`.

`matplotlib.mlab.rec_join(key, name, recs, jointype=u'outer', missing=0.0, postfixes=None)` Join a sequence of record arrays on single column key.
This function only joins a single column of the multiple record arrays

key is the column name that acts as a key
name is the name of the column that we want to join
recs is a list of record arrays to join
jointype is a string ‘inner’ or ‘outer’
missing is what any missing field is replaced by
postfixes if not None, a len recs sequence of postfixes

returns a record array with columns [rowkey, name0, name1, ... namen-1]. or if postfixes [PF0, PF1, ..., PFN-1] are supplied, [rowkey, namePF0, namePF1, ... namePFN-1].

Example:

```python
r = recs_join("date", "close", recs=[r0, r1], missing=0.)
```

matplotlib.mlab.\texttt{rk4}(\texttt{derivs}, y0, t)

Integrate 1D or ND system of ODEs using 4-th order Runge-Kutta. This is a toy implementation which may be useful if you find yourself stranded on a system w/o scipy. Otherwise use scipy.integrate().
y0 initial state vector
t sample times
derivs returns the derivative of the system and has the signature dy = derivs(yi, ti)

Example 1

```python
## 2D system
def derivs6(x, t):
    d1 = x[0] + 2*x[1]
    d2 = -3*x[0] + 4*x[1]
    return (d1, d2)
dt = 0.0005
t = arange(0.0, 2.0, dt)
y0 = (1,2)
yout = rk4(derivs6, y0, t)
```

Example 2:

```python
## 1D system
alpha = 2
def derivs(x, t):
    return -alpha*x + exp(-t)
y0 = 1
yout = rk4(derivs, y0, t)
```

If you have access to scipy, you should probably be using the scipy.integrate tools rather than this function.

matplotlib.mlab.\texttt{rms\_flat}(a)

Return the root mean square of all the elements of a, flattened out.
Matplotlib, Release 1.4.3

matplotlib.mlab.safe_isinf(x)
    numpy.isinf() for arbitrary types

matplotlib.mlab.safe_isnan(x)
    numpy.isnan() for arbitrary types

matplotlib.mlab.segments_intersect(s1, s2)
    Return True if s1 and s2 intersect. s1 and s2 are defined as:

    s1: (x1, y1), (x2, y2)
    s2: (x3, y3), (x4, y4)

matplotlib.mlab.slopes(x, y)
    slopes() calculates the slope y'(x)

    The slope is estimated using the slope obtained from that of a parabola through any three consecutive
    points.

    This method should be superior to that described in the appendix of A CONSISTENTLY WELL
    BEHAVED METHOD OF INTERPOLATION by Russel W. Stineman (Creative Computing July
    1980) in at least one aspect:

    Circles for interpolation demand a known aspect ratio between x- and y-values. For many
    functions, however, the abscissa are given in different dimensions, so an aspect ratio is
    completely arbitrary.

    The parabola method gives very similar results to the circle method for most regular cases but behaves
    much better in special cases.

    Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norber
    .Nemec at physik.uni-regensburg.de

    (inspired by a original implementation by Halldor Bjornsson, Icelandic Meteorological Office, March
    2006 halldor at vedur.is)

matplotlib.mlab.specgram(x, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None,
    cmap=None, xextent=None, pad_to=None, sides=None, scale_by_freq=None, mode=None)

    Compute a spectrogram.

    Call signature:

    specgram(x, NFFT=256, Fs=2, detrend=mlab.detrend_none,
             window=mlab.window_hanning, noverlap=128,
             cmap=None, xextent=None, pad_to=None, sides='default',
             scale_by_freq=None, mode='default')

    Compute and plot a spectrogram of data in x. Data are split into NFFT length segments and the
    spectrum of each section is computed. The windowing function window is applied to each segment,
    and the amount of overlap of each segment is specified with noverlap.

    x: 1-D array or sequence Array or sequence containing the data

    Keyword arguments:

    Fs: scalar The sampling frequency (samples per time unit). It is used to calculate the
         Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
window: callable or ndarray A function or a vector of length $NFFT$. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides: ['default' | 'onesided' | 'twosided'] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

pad_to: integer The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to fft(). The default is None, which sets pad_to equal to $NFFT$.

$NFFT$: integer The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

detrend: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or callable
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib is it a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

scale_by_freq: boolean
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

mode: ['default' | 'psd' | 'complex' | 'magnitude' | 'angle' | 'phase']
What sort of spectrum to use. Default is ‘psd’. which takes the power spectral density. ‘complex’ returns the complex-valued frequency spectrum. ‘magnitude’ returns the magnitude spectrum. ‘angle’ returns the phase spectrum without unwrapping. ‘phase’ returns the phase spectrum with unwrapping.

nooverlap: integer The number of points of overlap between blocks. The default value is 128.

Returns the tuple ($spectrum$, $freqs$, $t$):
$spectrum$: 2-D array columns are the periodograms of successive segments
$freqs$: 1-D array The frequencies corresponding to the rows in $spectrum$
$t$: 1-D array The times corresponding to midpoints of segments (i.e the columns in $spectrum$).

Note: detrend and scale_by_freq only apply when mode is set to ‘psd’

See also:
**psd() psd()** differs in the default overlap; in returning the mean of the segment periodograms; and in not returning times.

**complex_spectrum()** A single spectrum, similar to having a single segment when *mode* is ‘complex’.

**magnitude_spectrum()** A single spectrum, similar to having a single segment when *mode* is ‘magnitude’.

**angle_spectrum()** A single spectrum, similar to having a single segment when *mode* is ‘angle’.

**phase_spectrum()** A single spectrum, similar to having a single segment when *mode* is ‘phase’.

**matplotlib.mlab.stineman_interp(xi, x, y, yp=None)**

Given data vectors *x* and *y*, the slope vector *yp* and a new abscissa vector *xi*, the function **stineman_interp()** uses Stineman interpolation to calculate a vector *yi* corresponding to *xi*.

Here’s an example that generates a coarse sine curve, then interpolates over a finer abscissa:

```python
x = linspace(0,2*pi,20); y = sin(x); yp = cos(x)
xi = linspace(0,2*pi,40);
yi = stineman_interp(xi,x,y,yp);
plot(x,y, 'o',xi,yi)
```

The interpolation method is described in the article A CONSISTENTLY WELL BEHAVED METHOD OF INTERPOLATION by Russell W. Stineman. The article appeared in the July 1980 issue of Creative Computing with a note from the editor stating that while they were:

not an academic journal but once in a while something serious and original comes in
adding that this was “apparently a real solution” to a well known problem.

For *yp = None*, the routine automatically determines the slopes using the **slopes()** routine.

*x* is assumed to be sorted in increasing order.

For values *xi[j] < x[0]* or *xi[j] > x[-1]*, the routine tries an extrapolation. The relevance of the data obtained from this, of course, is questionable...

Original implementation by Halldor Bjornsson, Icelandic Meteorological Office, March 2006 halldor at vedur.is

Completely reworked and optimized for Python by Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec at physik.uni-regensburg.de

**matplotlib.mlab.stride_repeat(x, n, axis=0)**

Repeat the values in an array in a memory-efficient manner. Array *x* is stacked vertically *n* times.

**Warning:** It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

Call signature:

```python
stride_repeat(x, n, axis=0)
```

*x*: 1D array or sequence

Array or sequence containing the data.

*n*: integer
The number of time to repeat the array.

*axis*: integer
The axis along which the data will run.

**Refs:** stackoverflow: Repeat NumPy array without replicating data?

**matplotlib.mlab.stride_windows**(x, n, noverlap=None, axis=0)
Get all windows of x with length n as a single array, using strides to avoid data duplication.

**Warning:** It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

Call signature:

```
stride_windows(x, n, noverlap=0)
```

*x*: 1D array or sequence
Array or sequence containing the data.

*n*: integer
The number of data points in each window.

*noverlap*: integer
The overlap between adjacent windows.
Default is 0 (no overlap)

*axis*: integer
The axis along which the windows will run.

**Refs:** stackoverflow: Rolling window for 1D arrays in Numpy? stackoverflow: Using strides for an efficient moving average filter

**matplotlib.mlab.vector_lengths**(X, P=2.0, axis=None)
Finds the length of a set of vectors in n dimensions. This is like the numpy.norm() function for vectors, but has the ability to work over a particular axis of the supplied array or matrix.

Computes \( (\sum (x_i)^P)^{1/P} \) for each \( \{x_i\} \) being the elements of \( X \) along the given axis. If \( axis \) is None, compute over all elements of \( X \).

**matplotlib.mlab.window_hanning**(x)
Return x times the hanning window of len(x).

Call signature:

```
window_hanning(x)
```

**See also:**

`window_none()` `window_none()` is another window algorithm.

**matplotlib.mlab.window_none**(x)
No window function; simply return x.
Call signature:

window_none(x)

See also:

window_hanning() window_hanning() is another window algorithm.
59.1 matplotlib.offsetbox

The OffsetBox is a simple container artist. The child artist are meant to be drawn at a relative position to its parent. The [VH]Packer, DrawingArea and TextArea are derived from the OffsetBox.

The [VH]Packer automatically adjust the relative postions of their children, which should be instances of the OffsetBox. This is used to align similar artists together, e.g., in legend.

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The TextArea contains a single Text instance. The width and height of the TextArea instance is the width and height of the its child text.

```python
class matplotlib.offsetbox.AnchoredOffsetbox(loc,
                                          pad=0.4,
                                          borderpad=0.5,
                                          child=None,
                                          prop=None,
                                          frameon=True,
                                          bbox_to_anchor=None,
                                          bbox_transform=None,
                                          **kwargs)
```

An offset box placed according to the legend location loc. AnchoredOffsetbox has a single child. When multiple children is needed, use other OffsetBox class to enclose them. By default, the offset box is anchored against its parent axes. You may explicitly specify the bbox_to_anchor.

loc is a string or an integer specifying the legend location. The valid location codes are:

- 'upper right' : 1,
- 'upper left' : 2,
- 'lower left' : 3,
- 'lower right' : 4,
- 'right' : 5,
- 'center left' : 6,
- 'center right' : 7,
- 'lower center' : 8,
- 'upper center' : 9,
- 'center' : 10,

**pad** [pad around the child for drawing a frame. given in] fraction of fontsize.

**borderpad** : pad between offsetbox frame and the bbox_to_anchor,

**child** : OffsetBox instance that will be anchored.
prop : font property. This is only used as a reference for paddings.
frameon : draw a frame box if True.
bbox_to_anchor : bbox to anchor. Use self.axes.bbox if None.
bbox_transform : with which the bbox_to_anchor will be transformed.

draw(renderer)
    draw the artist

going_bbox_to_anchor()
    return the bbox that the legend will be anchored

going_child()
    return the child

going_children()
    return the list of children

going_extent(renderer)
    return the extent of the artist. The extent of the child added with the pad is returned

going_window_extent(renderer)
    get the bounding box in display space.

set_bbox_to_anchor(bbox, transform=None)
    set the bbox that the child will be anchored.
    bbox can be a Bbox instance, a list of [left, bottom, width, height], or a list of [left, bottom] where the width and height will be assumed to be zero. The bbox will be transformed to display coordinate by the given transform.

set_child(child)
    set the child to be anchored

update_frame(bbox, fontsize=None)

zorder = 5

class matplotlib.offsetbox.AnchoredText(s, loc, pad=0.4, borderpad=0.5, prop=None, **kwargs)
Bases: matplotlib.offsetbox.AnchoredOffsetbox
    AnchoredOffsetbox with Text.

Parameters s : string
    Text.
loc : str
    Location code.
    pad : float, optional
        Pad between the text and the frame as fraction of the font size.
borderpad : float, optional
    Pad between the frame and the axes (or bbox_to_anchor).
prop : matplotlib.font_manager.FontProperties
Font properties.

Notes

Other keyword parameters of AnchoredOffsetbox are also allowed.

class matplotlib.offsetbox.AnnotationBbox

An annotation-like class, but with offset box instead of Text.

offsetbox : OffsetBox instance

xycoords [same as Annotation but can be a tuple of two] strings which are interpreted as x and y coordinates.

boxcoords [similar to textcoords as Annotation but can be a tuple of two strings which are interpreted as x and y coordinates.

box_alignment [a tuple of two floats for a vertical and] horizontal alignment of the offset box w.r.t. the boxcoords. The lower-left corner is (0.0) and upper-right corner is (1.1). other parameters are identical to that of Annotation.

contains(event)

draw(renderer)

Draw the Annotation object to the given renderer.

get_children()

get_fontsize(s=None)

return font size in points

set_figure(fig)

set_fontsize(s=None)

set font size in points

update_positions(renderer)

Update the pixel positions of the annotated point and the text.

xyann

zorder = 3
class matplotlib.offsetbox.AuxTransformBox(aux_transform)
   Bases: matplotlib.offsetbox.OffsetBox

   Offset Box with the aux_transform. Its children will be transformed with the aux_transform first then
   will be offseted. The absolute coordinate of the aux_transform is meaning as it will be automatically
   adjust so that the left-lower corner of the bounding box of children will be set to (0,0) before the offset
   transform.

   It is similar to drawing area, except that the extent of the box is not predetermined but calculated from
   the window extent of its children. Furthermore, the extent of the children will be calculated in the
   transformed coordinate.

   add_artist(a)
      Add any Artist to the container box

   draw(renderer)
      Draw the children

   get_extent(renderer)

   get_offset()
      return offset of the container.

   get_transform()
      Return the Transform applied to the children

   get_window_extent(renderer)
      get the bounding box in display space.

   set_offset(xy)
      set offset of the container.

      Accept : tuple of x,y coordinate in disokay units.

   set_transform(t)
      set_transform is ignored.

class matplotlib.offsetbox.DraggableAnnotation((annotation, use_blit=False)
   Bases: matplotlib.offsetbox.DraggableBase

   finalize_offset()

   save_offset()

   update_offset(dx, dy)

class matplotlib.offsetbox.DraggableBase(ref_artist, use_blit=False)
   Bases: object

   helper code for a draggable artist (legend, offsetbox) The derived class must override following two
   method.

   def saveoffset(self): pass
def update_offset(self, dx, dy): pass

saveoffset is called when the object is picked for dragging and it is meant to save reference position of the artist.

update_offset is called during the dragging. dx and dy is the pixel offset from the point where the mouse drag started.

Optionally you may override following two methods.

def artist_picker(self, artist, evt): return self.ref_artist.contains(evt)
def finalize_offset(self): pass

artist_picker is a picker method that will be used. finalize_offset is called when the mouse is released. In current implementation of DraggableLegend and DraggableAnnotation, update_offset places the artists simply in display coordinates. And finalize_offset recalculate their position in the normalized axes coordinate and set a relevant attribute.

artist_picker(artist, evt)

disconnect()

disconnect the callbacks

finalize_offset()

on_motion(evt)

on_motion_blit(evt)

on_pick(evt)

on_release(event)

save_offset()

update_offset(dx, dy)

class matplotlib.offsetbox.DraggableOffsetBox(ref_artist, offsetbox, use_blit=False)
Bases: matplotlib.offsetbox.DraggableBase

get_loc_in_canvas()

save_offset()

update_offset(dx, dy)
class matplotlib.offsetbox.DrawingArea(width, height, xdescent=0.0, ydescent=0.0, clip=True)

Bases: matplotlib.offsetbox.OffsetBox

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed.

width, height : width and height of the container box. xdescent, ydescent : descent of the box in x- and y-direction.

add_artist(a)
    Add any Artist to the container box

draw(renderer)
    Draw the children

get_extent(renderer)
    Return with, height, xdescent, ydescent of box

get_offset()
    return offset of the container.

get_transform()
    Return the Transform applied to the children

get_window_extent(renderer)
    get the bounding box in display space.

set_offset(xy)
    set offset of the container.

    Accept : tuple of x,y coordinate in disokay units.

set_transform(t)
    set_transform is ignored.

class matplotlib.offsetbox.HPacker(pad=None, sep=None, width=None, height=None, align=u'baseline', mode=u'fixed', children=None)

Bases: matplotlib.offsetbox.PackerBase

The HPacker has its children packed horizontally. It automatically adjusts the relative positions of children at draw time.

Parameters pad : float, optional
    Boundary pad.

sep : float, optional
    Spacing between items.

width : float, optional

height : float, optional
    Width and height of the container box, calculated if None.

align : str
    Alignment of boxes.

mode : str
    Packing mode.
Notes

`pad` and `sep` need to be given in points and will be scaled with the renderer dpi, while `width` and `height` need to be in pixels.

**get_extent_offsets**(renderer)
update offset of children and return the extents of the box

**class** matplotlib.offsetbox.OffsetBox(*args, **kwargs)
Bases: matplotlib.artist.Artist

The OffsetBox is a simple container artist. The child artist are meant to be drawn at a relative position to its parent.

**contains**(mouseevent)

**draw**(renderer)
Update the location of children if necessary and draw them to the given renderer.

**get_children**()
Return a list of artists it contains.

**get_extent**(renderer)
Return with, height, xdescent, ydescent of box

**get_extent_offsets**(renderer)

**get_offset**(width, height, xdescent, ydescent, renderer)
Get the offset accepts extent of the box

**get_visible_children**()
Return a list of visible artists it contains.

**get_window_extent**(renderer)
get the bounding box in display space.

**set_figure**(fig)
Set the figure accepts a class:Figure instance

**set_height**(height)
Set the height accepts float

**set_offset**(xy)
Set the offset accepts x, y, tuple, or a callable object.

**set_width**(width)
Set the width
accepts float

**class** matplotlib.offsetbox.OffsetImage**(arr, zoom=1, cmap=None, norm=None, interpolation=None, origin=None, filternorm=1, filterrad=4.0, resample=False, dpi_cor=True, **kwargs)**  
Bases: matplotlib.offsetbox.OffsetBox

draw**(renderer)**  
Draw the children

get_children()

get_data()

get_extent**(renderer)**

get_offset()  
return offset of the container.

get_window_extent**(renderer)**  
get the bounding box in display space.

get_zoom()

set_data**(arr)**

set_zoom**(zoom)**

**class** matplotlib.offsetbox.PackerBase**(pad=None, sep=None, width=None, height=None, align=None, mode=None, children=None)**  
Bases: matplotlib.offsetbox.OffsetBox

**Parameters** pad : float, optional  
Boundary pad.

sep : float, optional  
Spacing between items.

width : float, optional

height : float, optional  
Width and height of the container box, calculated if None.

align : str, optional  
Alignment of boxes. Can be one of top, bottom, left, right, center and baseline

mode : str, optional  
Packing mode.
Notes

`pad` and `sep` need to be given in points and will be scaled with the renderer dpi, while `width` and `height` need to be in pixels.

class matplotlib.offsetbox.PaddedBox(child, pad=None, draw_frame=False, patch_attr=None)

Bases: matplotlib.offsetbox.OffsetBox

`pad`: boundary pad

**Note**: `pad` need to be given in points and will be scaled with the renderer dpi, while `width` and `height` need to be in pixels.

draw(renderer)

Update the location of children if necessary and draw them to the given renderer.

draw_frame(renderer)

get_extent_offsets(renderer)

update offset of children and return the extents of the box

update_frame(bbox, fontsize=None)

class matplotlib.offsetbox.TextArea(s, textprops=None, multilinebaseline=None, minimumdescent=True)

Bases: matplotlib.offsetbox.OffsetBox

The TextArea contains a single Text instance. The text is placed at (0,0) with baseline+left alignment. The width and height of the TextArea instance is the width and height of its child text.

Parameters s: str

- a string to be displayed.
- textprops: FontProperties, optional
- multilinebaseline: bool, optional
- If True, baseline for multline text is adjusted so that it is (approximately) center-aligned with singleline text.
- minimumdescent: bool, optional
- If True, the box has a minimum descent of “p”.

draw(renderer)

Draw the children

get_extent(renderer)

get_minimumdescent()

get minimumdescent.

get_multilinebaseline()

get multilinebaseline.
get_offset()  
return offset of the container.

get_text()  
get text

get_window_extent(renderer)  
get the bounding box in display space.

set_minimumdescent(t)  
Set minimumdescent.

If True, extent of the single line text is adjusted so that it has minimum descent of “p”

set_multilinebaseline(t)  
Set multilinebaseline.

If True, baseline for multilime text is adjusted so that it is (approximately) center-aligned with singleline text.

set_offset(xy)  
set offset of the container.

Accept : tuple of x,y coordinates in display units.

set_text(s)  
set text

set_transform(t)  
set_transform is ignored.

class matplotlib.offsetbox.VPacker(pad=None, sep=None, width=None, height=None,  
align=u'baseline', mode=u'fixed', children=None)  
Bases: matplotlib.offsetbox.PackerBase

The VPacker has its children packed vertically. It automatically adjust the relative positions of children in the drawing time.

Parameters pad : float, optional  
Boundary pad.

sep : float, optional  
Spacing between items.

width : float, optional

height : float, optional  
width and height of the container box, calculated if None.

align : str, optional  
Alignment of boxes.

mode : str, optional  
Packing mode.
Notes

*pad* and *sep* need to be given in points and will be scaled with the renderer dpi, while *width* and *height* need to be in pixels.

**get_extent_offsets**(renderer)

update offset of childrens and return the extents of the box

matplotlib.offsetbox.bbox_artist(*args, **kwargs)
60.1 matplotlib.patches

```python
class matplotlib.patches.Arc(xy, width, height, angle=0.0, theta1=0.0, theta2=360.0, **kwargs)
Bases: matplotlib.patches.Ellipse
```

An elliptical arc. Because it performs various optimizations, it can not be filled.

The arc must be used in an Axes instance—it can not be added directly to a Figure—because it is optimized to only render the segments that are inside the axes bounding box with high resolution.

The following args are supported:

- **xy** center of ellipse
- **width** length of horizontal axis
- **height** length of vertical axis
- **angle** rotation in degrees (anti-clockwise)
- **theta1** starting angle of the arc in degrees
- **theta2** ending angle of the arc in degrees

If `theta1` and `theta2` are not provided, the arc will form a complete ellipse.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgcolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[’miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[’solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

draw\((\text{artist, renderer, *args, **kwargs})\)

Ellipses are normally drawn using an approximation that uses eight cubic bezier splines. The error of this approximation is $1.89818e-6$, according to this unverified source:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.


There is a use case where very large ellipses must be drawn with very high accuracy, and it is too expensive to render the entire ellipse with enough segments (either splines or line segments). Therefore, in the case where either radius of the ellipse is large enough that the error of the spline approximation will be visible (greater than one pixel offset from the ideal), a different technique is used.

In that case, only the visible parts of the ellipse are drawn, with each visible arc using a fixed number of spline segments (8). The algorithm proceeds as follows:

1. The points where the ellipse intersects the axes bounding box are located. (This is done by performing an inverse transformation on the axes bbox such that it is relative to the unit circle – this makes the intersection calculation much easier than doing rotated ellipse intersection directly).

This uses the “line intersecting a circle” algorithm from:


2. The angles of each of the intersection points are calculated.

3. Proceeding counterclockwise starting in the positive x-direction, each of the visible arc-segments between the pairs of vertices are drawn using the bezier arc approximation technique implemented in matplotlib.path.Path.arc().
class matplotlib.patches.Arrow(x, y, dx, dy, width=1.0, **kwargs)
Bases: matplotlib.patches.Patch

An arrow patch.

Draws an arrow, starting at (x, y), direction and length given by (dx, dy) the width of the arrow is scaled by width.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt’</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_patch_transform()

get_path()
class matplotlib.patches.ArrowStyle

Bases: matplotlib.patches._Style

ArrowStyle is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path. These are mainly used with FancyArrowPatch.

A arrowstyle object can be either created as:

ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)

or:

ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)

or:

ArrowStyle("Fancy, head_length=.4, head_width=.4, tail_width=.4")

The following classes are defined:

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve</td>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>CurveB</td>
<td>-&gt;</td>
<td>head_length=0.4, head_width=0.2</td>
</tr>
<tr>
<td>BracketB</td>
<td>[-]</td>
<td>widthB=1.0, lengthB=0.2, angleB=None</td>
</tr>
<tr>
<td>Curve-</td>
<td>-&gt;</td>
<td></td>
</tr>
<tr>
<td>FilledB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CurveA</td>
<td>&lt;-</td>
<td>head_length=0.4, head_width=0.2</td>
</tr>
<tr>
<td>CurveAB</td>
<td>&lt;-&gt;</td>
<td>head_length=0.4, head_width=0.2</td>
</tr>
<tr>
<td>Curve-</td>
<td>&lt;</td>
<td>--</td>
</tr>
<tr>
<td>FilledA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curve-</td>
<td>&lt;</td>
<td>--</td>
</tr>
<tr>
<td>FilledAB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BracketA</td>
<td>]-</td>
<td>widthA=1.0, lengthA=0.2, angleA=None</td>
</tr>
<tr>
<td>BracketAB</td>
<td>]-[</td>
<td>widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None</td>
</tr>
<tr>
<td>Fancy</td>
<td>fancy</td>
<td>head_length=0.4, head_width=0.4, tail_width=0.4</td>
</tr>
<tr>
<td>Simple</td>
<td>simple</td>
<td>head_length=0.5, head_width=0.5, tail_width=0.2</td>
</tr>
<tr>
<td>Wedge</td>
<td>wedge</td>
<td>tail_width=0.3, shrink_factor=0.5</td>
</tr>
<tr>
<td>BarAB</td>
<td>[ ]</td>
<td>widthA=1.0, angleA=None, widthB=1.0, angleB=None</td>
</tr>
</tbody>
</table>

An instance of any arrow style class is an callable object, whose call signature is:

```
__call__(self, path, mutation_size, linewidth, aspect_ratio=1.)
```

and it returns a tuple of a Path instance and a boolean value. path is a Path instance along which the arrow will be drawn. mutation_size and aspect_ratio has a same meaning as in BoxStyle. linewidth is a line width to be stroked. This is meant to be used to correct the location of the head so that it does not overshoot the destination point, but not all classes support it.
class BarAB(widthA=1.0, angleA=None, widthB=1.0, angleB=None)
    Bases: matplotlib.patches._Bracket
    An arrow with a bar(|) at both ends.
    widthA width of the bracket
    lengthA length of the bracket
    angleA angle between the bracket and the line
    widthB width of the bracket
    lengthB length of the bracket
    angleB angle between the bracket and the line

class ArrowStyle.BracketA(widthA=1.0, lengthA=0.2, angleA=None)
    Bases: matplotlib.patches._Bracket
    An arrow with a bracket(]) at its end.
    widthA width of the bracket
    lengthA length of the bracket
    angleA angle between the bracket and the line

class ArrowStyle.BracketAB(widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0,
                            lengthB=0.2, angleB=None)
    Bases: matplotlib.patches._Bracket
An arrow with a bracket([]) at both ends.

- **widthA** width of the bracket
- **lengthA** length of the bracket
- **angleA** angle between the bracket and the line
- **widthB** width of the bracket
- **lengthB** length of the bracket
- **angleB** angle between the bracket and the line

```python
class ArrowStyle.BracketB(widthB=1.0, lengthB=0.2, angleB=None)
    Bases: matplotlib.patches._Bracket
```

An arrow with a bracket([]) at its end.

- **widthB** width of the bracket
- **lengthB** length of the bracket
- **angleB** angle between the bracket and the line

```python
class ArrowStyle.Curve
    Bases: matplotlib.patches._Curve
```

A simple curve without any arrow head.

```python
class ArrowStyle.CurveA(head_length=0.4, head_width=0.2)
    Bases: matplotlib.patches._Curve
```

An arrow with a head at its begin point.

- **head_length** length of the arrow head
- **head_width** width of the arrow head

```python
class ArrowStyle.CurveAB(head_length=0.4, head_width=0.2)
    Bases: matplotlib.patches._Curve
```

An arrow with heads both at the begin and the end point.

- **head_length** length of the arrow head
- **head_width** width of the arrow head

```python
class ArrowStyle.CurveB(head_length=0.4, head_width=0.2)
    Bases: matplotlib.patches._Curve
```

An arrow with a head at its end point.

- **head_length** length of the arrow head
- **head_width** width of the arrow head

```python
class ArrowStyle.CurveFilledA(head_length=0.4, head_width=0.2)
    Bases: matplotlib.patches._Curve
```

An arrow with filled triangle head at the begin.

- **head_length** length of the arrow head
- **head_width** width of the arrow head

```python
class ArrowStyle.CurveFilledAB(head_length=0.4, head_width=0.2)
    Bases: matplotlib.patches._Curve
```

An arrow with filled triangle heads both at the begin and the end point.

- **head_length** length of the arrow head
- **head_width** width of the arrow head
class ArrowStyle.CurveFilledB(head_length=0.4, head_width=0.2)
    Bases: matplotlib.patches._Curve

    An arrow with filled triangle head at the end.
    head_length length of the arrow head
    head_width width of the arrow head

class ArrowStyle.Fancy(head_length=0.4, head_width=0.4, tail_width=0.4)
    Bases: matplotlib.patches._Base

    A fancy arrow. Only works with a quadratic bezier curve.
    head_length length of the arrow head
    head_with width of the arrow head
    tail_width width of the arrow tail
    transmute(path, mutation_size, linewidth)

class ArrowStyle.Simple(head_length=0.5, head_width=0.5, tail_width=0.2)
    Bases: matplotlib.patches._Base

    A simple arrow. Only works with a quadratic bezier curve.
    head_length length of the arrow head
    head_with width of the arrow head
    tail_width width of the arrow tail
    transmute(path, mutation_size, linewidth)

class ArrowStyle.Wedge(tail_width=0.3, shrink_factor=0.5)
    Bases: matplotlib.patches._Base

    Wedge(?) shape. Only works with a quadratic bezier curve. The begin point has a
    width of the tail_width and the end point has a width of 0. At the middle, the width is
    shrink_factor*tail_width.
    tail_width width of the tail
    shrink_factor fraction of the arrow width at the middle point
    transmute(path, mutation_size, linewidth)

class matplotlib.patches.BoxStyle
    Bases: matplotlib.patches._Style

    BoxStyle is a container class which defines several boxstyle classes, which are used for
    FancyBoxPatch.

    A style object can be created as:

    BoxStyle.Round(pad=0.2)

    or:

    BoxStyle("Round", pad=0.2)

    or:
BoxStyle("Round, pad=0.2")

Following boxstyle classes are defined.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>circle</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>

An instance of any boxstyle class is an callable object, whose call signature is:

```
__call__(self, x0, y0, width, height, mutation_size, aspect_ratio=1.)
```

and returns a Path instance. `x0, y0, width` and `height` specify the location and size of the box to be drawn. `mutation_scale` determines the overall size of the mutation (by which I mean the transformation of the rectangle to the fancy box). `mutation_aspect` determines the aspect-ratio of the mutation.
A simple circle box.

**Parameters**
- `pad` : float
  The amount of padding around the original box.

**transmute** $(x0, y0, width, height, mutation_size)$

A simple circle box.

**BoxStyle.LArrow**

Bases: `matplotlib.patches._Base`

(Left) Arrow Box

**transmute** $(x0, y0, width, height, mutation_size)$
(right) Arrow Box

\texttt{transmute}(x0, y0, width, height, mutation_size)

class \texttt{BoxStyle.Round}(pad=0.3, rounding_size=None)

Bases: \texttt{matplotlib.patches._Base}

A box with round corners.

\textit{pad} amount of padding

\textit{rounding_size} rounding radius of corners. \textit{pad} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class \texttt{BoxStyle.Round4}(pad=0.3, rounding_size=None)

Bases: \texttt{matplotlib.patches._Base}

Another box with round edges.

\textit{pad} amount of padding

\textit{rounding_size} rounding size of edges. \textit{pad} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class \texttt{BoxStyle.Roundtooth}(pad=0.3, tooth_size=None)

Bases: \texttt{matplotlib.patches.Sawtooth}

A rounded tooth box.

\textit{pad} amount of padding

\textit{tooth_size} size of the sawtooth. \textit{pad*} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class \texttt{BoxStyle.Sawtooth}(pad=0.3, tooth_size=None)

Bases: \texttt{matplotlib.patches._Base}

A sawtooth box.

\textit{pad} amount of padding

\textit{tooth_size} size of the sawtooth. \textit{pad*} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class \texttt{BoxStyle.Square}(pad=0.3)

Bases: \texttt{matplotlib.patches._Base}

A simple square box.

\textit{pad} amount of padding

\texttt{transmute}(x0, y0, width, height, mutation_size)

class \texttt{matplotlib.patches.Circle}(xy, radius=5, **kwargs)

Bases: \texttt{matplotlib.patches.Ellipse}

A circle patch.
Create true circle at center \( xy = (x, y) \) with given \( radius \). Unlike \texttt{CirclePolygon} which is a polygonal approximation, this uses Bézier splines and is much closer to a scale-free circle.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an \texttt{Axes} instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a \texttt{matplotlib.transforms.Bbox} instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(\texttt{Path, Transform}), \texttt{Patch}, None]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a \texttt{matplotlib.figure.Figure} instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>\texttt{Transform} instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

\texttt{get_radius()} 
return the radius of the circle

\texttt{radius} 
return the radius of the circle

\texttt{set_radius(radius)}
Set the radius of the circle

ACCEPTS: float
class matplotlib.patches.CirclePolygon(xy, radius=5, resolution=20, **kwargs)
Bases: matplotlib.patches.RegularPolygon

A polygon-approximation of a circle patch.

Create a circle at $xy = (x, y)$ with given radius. This circle is approximated by a regular polygon with
resolution sides. For a smoother circle drawn with splines, see Circle.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with %s conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
class matplotlib.patches.ConnectionPatch(xyA, xyB, coordsA, coordsB=None, axesA=None, axesB=None, arrowstyle='-', arrow_transmuter=None, connectionstyle='arc3', connector=None, patchA=None, patchB=None, shrinkA=0.0, shrinkB=0.0, mutation_scale=10.0, mutation_aspect=None, clip_on=False, dpi_cor=1.0, **kwargs)

Bases: matplotlib.patches.FancyArrowPatch

A ConnectionPatch class is to make connecting lines between two points (possibly in different axes).

Connect point xyA in coordsA with point xyB in coordsB

Valid keys are

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
</tbody>
</table>

Any key for matplotlib.patches.PathPatch

coordsA and coordsB are strings that indicate the coordinates of xyA and xyB.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>0,0 is lower left of figure and 1,1 is upper, right</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>0,1 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>‘offset points’</td>
<td>Specify an offset (in points) from the xy value</td>
</tr>
<tr>
<td>‘polar’</td>
<td>you can specify theta, r for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native “data” coordinate system.</td>
</tr>
</tbody>
</table>

draw(renderer)
   Draw.

get_annotation_clip()
   Return annotation_clip attribute. See set_annotation_clip() for the meaning of return values.

get_path_in_displaycoord()
   Return the mutated path of the arrow in the display coord

set_annotation_clip(b)
   set annotation_clip attribute.
   • True: the annotation will only be drawn when self.xy is inside the axes.
   • False: the annotation will always be drawn regardless of its position.
   • None: the self.xy will be checked only if xycoords is “data”

class matplotlib.patches.ConnectionStyle
   Bases: matplotlib.patches._Style

   ConnectionStyle is a container class which defines several connectionstyle classes, which is used to create a path between two points. These are mainly used with FancyArrowPatch.

   A connectionstyle object can be either created as:

   ConnectionStyle.Arc3(rad=0.2)

   or:
ConnectionStyle("Arc3", rad=0.2)

or:

ConnectionStyle("Arc3, rad=0.2")

The following classes are defined

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle</td>
<td>angle</td>
<td>angleA=90,angleB=0,rad=0.0</td>
</tr>
<tr>
<td>Angle3</td>
<td>angle3</td>
<td>angleA=90,angleB=0</td>
</tr>
<tr>
<td>Arc</td>
<td>arc</td>
<td>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</td>
</tr>
<tr>
<td>Arc3</td>
<td>arc3</td>
<td>rad=0.0</td>
</tr>
<tr>
<td>Bar</td>
<td>bar</td>
<td>armA=0.0,armB=0.0,fraction=0.3,angle=None</td>
</tr>
</tbody>
</table>

An instance of any connection style class is an callable object, whose call signature is:

```
__call__(self, posA, posB, patchA=None, patchB=None, shrinkA=2., shrinkB=2.)
```

and it returns a Path instance. posA and posB are tuples of x,y coordinates of the two points to be connected. patchA (or patchB) is given, the returned path is clipped so that it start (or end) from the boundary of the patch. The path is further shrunk by shrinkA (or shrinkB) which is given in points.

class Angle(angleA=90, angleB=0, rad=0.0)

    Bases: matplotlib.patches._Base

    Creates a picewise continuous quadratic bezier path between two points. The path has a one passing-through point placed at the intersecting point of two lines which crosses the start (or end) point and has a angle of angleA (or angleB). The connecting edges are rounded with rad.

    *angleA* starting angle of the path

    *angleB* ending angle of the path

    *rad* rounding radius of the edge

    connect(posA, posB)

class ConnectionStyle.Angle3(angleA=90, angleB=0)

    Bases: matplotlib.patches._Base

    Creates a simple quadratic bezier curve between two points. The middle control points is placed at the intersecting point of two lines which crosses the start (or end) point and has a angle of angleA (or angleB).

    *angleA* starting angle of the path

    *angleB* ending angle of the path

    connect(posA, posB)

class ConnectionStyle.Arc(angleA=0, angleB=0, armA=None, armB=None, rad=0.0)

    Bases: matplotlib.patches._Base
Creates a picewise continuous quadratic bezier path between two points. The path can have two passing-through points, a point placed at the distance of armA and angle of angleA from point A, another point with respect to point B. The edges are rounded with rad.

angleA : starting angle of the path
angleB : ending angle of the path
armA : length of the starting arm
armB : length of the ending arm
rad : rounding radius of the edges

class ConnectionStyle.Arc3(rad=0.0)
Bases: matplotlib.patches._Base

Creates a simple quadratic bezier curve between two points. The curve is created so that the middle control points (C1) is located at the same distance from the start (C0) and end points (C2) and the distance of the C1 to the line connecting C0-C2 is rad times the distance of C0-C2.

rad : curvature of the curve.

class ConnectionStyle.Bar(armA=0.0, armB=0.0, fraction=0.3, angle=None)
Bases: matplotlib.patches._Base

A line with angle between A and B with armA and armB. One of the arm is extend so that they are connected in a right angle. The length of armA is determined by (armA + fraction x AB distance). Same for armB.

armA : minimum length of armA
armB : minimum length of armB
fraction : [a fraction of the distance between two points that] will be added to armA and armB.
angle : [angle of the connecting line (if None, parallel to A] and B)

class matplotlib.patches.Ellipse(xy, width, height, angle=0.0, **kwargs)
Bases: matplotlib.patches.Patch

A scale-free ellipse.

xy : center of ellipse
width : total length (diameter) of horizontal axis
height : total length (diameter) of vertical axis
angle : rotation in degrees (anti-clockwise)

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 60.5 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with 's' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

`contains(ev)`

`get_patch_transform()`

`get_path()`
Return the vertices of the rectangle

`class matplotlib.patches.FancyArrow(x, y, dx, dy, width=0.001, length_includes_head=False, head_width=None, head_length=None, shape=u'full', overhang=0, head_starts_at_zero=False, **kwargs)`

Bases: matplotlib.patches.Polygon

Like Arrow, but lets you set head width and head height independently.

**Constructor arguments**

- `width`: float (default: 0.001) width of full arrow tail
**length_includes_head**: [True | False] (default: False) True if head is to be counted in calculating the length.

**head_width**: float or None (default: 3*width) total width of the full arrow head

**head_length**: float or None (default: 1.5 * head_width) length of arrow head

**shape**: ['full', 'left', 'right'] (default: ‘full’) draw the left-half, right-half, or full arrow

**overhang**: float (default: 0) fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

**head_starts_at_zero**: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from Patch) are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgescolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
class matplotlib.patches.FancyArrowPatch(posA=None, posB=None, path=None, arrowstyle=u'simple', arrow_transmuter=None, connectionstyle=u'arc3', connector=None, patchA=None, patchB=None, shrinkA=2.0, shrinkB=2.0, mutation_scale=1.0, mutation_aspect=None, dpi_cor=1.0, **kwargs)

Bases: matplotlib.patches.Patch

A fancy arrow patch. It draws an arrow using the :class:ArrowStyle.

If posA and posB is given, a path connecting two point are created according to the connectionstyle. The path will be clipped with patchA and patchB and further shrinked by shrinkA and shrinkB. An arrow is drawn along this resulting path using the arrowstyle parameter. If path provided, an arrow is drawn along this path and patchA, patchB, shrinkA, and shrinkB are ignored.

The connectionstyle describes how posA and posB are connected. It can be an instance of the ConnectionStyle class (matplotlib.patches.ConnectionStyle) or a string of the connectionstyle name, with optional comma-separated attributes. The following connection styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle</td>
<td>angle</td>
<td>angleA=90,angleB=0,rad=0.0</td>
</tr>
<tr>
<td>Angle3</td>
<td>angle3</td>
<td>angleA=90,angleB=0</td>
</tr>
<tr>
<td>Arc</td>
<td>arc</td>
<td>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</td>
</tr>
<tr>
<td>Arc3</td>
<td>arc3</td>
<td>rad=0.0</td>
</tr>
<tr>
<td>Bar</td>
<td>bar</td>
<td>armA=0.0,armB=0.0,fraction=0.3,angle=None</td>
</tr>
</tbody>
</table>

The arrowstyle describes how the fancy arrow will be drawn. It can be string of the available arrowstyle names, with optional comma-separated attributes, or one of the ArrowStyle instance. The optional attributes are meant to be scaled with the mutation_scale. The following arrow styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve</td>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>CurveB</td>
<td>-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>BracketB</td>
<td>[-</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Curve-</td>
<td>-</td>
<td>&gt;</td>
</tr>
<tr>
<td>FilledB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CurveA</td>
<td>&lt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveAB</td>
<td>&lt;-</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>Curve-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FilledA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curve-</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>FilledAB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BracketA</td>
<td>]-</td>
<td>widthA=1.0,lengthA=0.2,angleA=None</td>
</tr>
<tr>
<td>BracketAB</td>
<td>]-</td>
<td>widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Fancy</td>
<td>fancy</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>Simple</td>
<td>simple</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>Wedge</td>
<td>wedge</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
<tr>
<td>BarAB</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

mutation_scale [a value with which attributes of arrowstyle] (e.g., head_length) will be scaled. default=1.
mutation_aspect [The height of the rectangle will be] squeezed by this value before the mutation and
the mutated box will be stretched by the inverse of it. default=None.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/']</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

draw(renderer)

get_arrowstyle()  
Return the arrowstyle object

get_connectionstyle()  
Return the ConnectionStyle instance

get_dpi_cor()  
dpi_cor is currently used for linewidth-related things and shink factor. Mutation scale is not affected by this.
get_mutation_aspect()  
Return the aspect ratio of the bbox mutation.

get_mutation_scale()  
Return the mutation scale.

get_path()  
return the path of the arrow in the data coordinate. Use get_path_in_displaycoord() method to retrieve the arrow path in the display coord.

get_path_in_displaycoord()  
Return the mutated path of the arrow in the display coord

set_arrowstyle(arrowstyle=None, **kw)  
Set the arrow style.  
arrowstyle can be a string with arrowstyle name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords.  
set_arrowstyle("Fancy,head_length=.2") set_arrowstyle("fancy", head_length=0.2)  
Old attrs simply are forgotten.  
Without argument (or with arrowstyle= None), return available box styles as a list of strings.

set_connectionstyle(connectionstyle, **kw)  
Set the connection style.  
connectionstyle can be a string with connectionstyle name with optional comma-separated attributes. Alternatively, the attrs can be proibded as keywords.  
set_connectionstyle("arc,angleA=0,armA=30,rad=10") set_connectionstyle("arc", angleA=0,armA=30,rad=10)  
Old attrs simply are forgotten.  
Without argument (or with connectionstyle= None), return available styles as a list of strings.

set_dpi_cor(dpi_cor)  
dpi_cor is currently used for linewidth-related things and shink factor. Mutation scale is not affected by this.

set_mutation_aspect(aspect)  
Set the aspect ratio of the bbox mutation.  
ACCEPTS: float

set_mutation_scale(scale)  
Set the mutation scale.  
ACCEPTS: float

set_patchA(patchA)  
set the begin patch.

set_patchB(patchB)  
set the begin patch

set_positions(posA, posB)  
set the begin end end positions of the connecting path. Use current vlaue if None.
class `matplotlib.patches.FancyBboxPatch`(xy, width, height, boxstyle='round', bbox_transmuter=None, mutation_scale=1.0, mutation_aspect=None, **kwargs)

Bases: `matplotlib.patches.Patch`

Draw a fancy box around a rectangle with lower left at `xy` = (*x, y) with specified width and height.

`FancyBboxPatch` class is similar to `Rectangle` class, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the `BoxTransmuterBase` and its derived classes.

`xy` = lower left corner

`width`, `height`

`boxstyle` determines what kind of fancy box will be drawn. It can be a string of the style name with a comma separated attribute, or an instance of `BoxStyle`. Following box styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>circle</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>

`mutation_scale`: a value with which attributes of boxstyle (e.g., pad) will be scaled. default=1.

`mutation_aspect`: The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. default=None.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/||</td>
</tr>
</tbody>
</table>
Table 60.8 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid', 'dashed', 'dashdot', 'dotted']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True, False]</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_bbox()

get_boxstyle()
Return the boxstyle object

get_height()
Return the height of the rectangle

get_mutation_aspect()
Return the aspect ratio of the bbox mutation.

get_mutation_scale()
Return the mutation scale.

get_path()
Return the mutated path of the rectangle

get_width()
Return the width of the rectangle

get_x()
Return the left coord of the rectangle

get_y()
Return the bottom coord of the rectangle

set_bounds(*args)
Set the bounds of the rectangle: l,b,w,h

    ACCEPTS: (left, bottom, width, height)

set_boxstyle(boxstyle=None, **kw)
Set the box style.
boxstyle can be a string with boxstyle name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords:

```python
set_boxstyle("round, pad=0.2")
set_boxstyle("round", pad=0.2)
```

Old attrs simply are forgotten.

Without argument (or with boxstyle = None), it returns available box styles.

```python
set_height(h)
Set the height rectangle
ACCEPTS: float
```

```python
set_mutation_aspect(aspect)
Set the aspect ratio of the bbox mutation.
ACCEPTS: float
```

```python
set_mutation_scale(scale)
Set the mutation scale.
ACCEPTS: float
```

```python
set_width(w)
Set the width rectangle
ACCEPTS: float
```

```python
set_x(x)
Set the left coord of the rectangle
ACCEPTS: float
```

```python
set_y(y)
Set the bottom coord of the rectangle
ACCEPTS: float
```

```python
class matplotlib.patches.Patch(edgecolor=None, facecolor=None, color=None, linewidth=None, linestyle=None, antialiased=None, hatch=None, fill=True, capstyle=None, joinstyle=None, **kwargs)
```

Bases: matplotlib.artist.Artist

A patch is a 2D artist with a face color and an edge color.

If any of edgecolor, facecolor, linewidth, or antialiased are None, they default to their rc params setting.

The following kwarg properties are supported

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 60.9 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>([Path, Transform]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**contains** *(mouseevent, radius=None)*

Test whether the mouse event occurred in the patch.

Returns T/F, {}.

**contains_point** *(point, radius=None)*

Returns True if the given point is inside the path (transformed with its transform attribute).

**draw** *(artist, renderer, *args, **kwargs)*

Draw the Patch to the given renderer.

**fill**

return whether fill is set

**get_aa()**

Returns True if the Patch is to be drawn with antialiasing.
get_antialiased()
    Returns True if the Patch is to be drawn with antialiasing.

get_capstyle()
    Return the current capstyle

get_data_transform()
    Return the Transform instance which maps data coordinates to physical coordinates.

get_ec()
    Return the edge color of the Patch.

g_getedgcolor()
    Return the edge color of the Patch.

get_extents()
    Return a Bbox object defining the axis-aligned extents of the Patch.

get_facecolor()
    Return the face color of the Patch.

get_fc()
    Return the face color of the Patch.

get_fill()
    return whether fill is set

get_hatch()
    Return the current hatching pattern

get_joinstyle()
    Return the current joinstyle

get_linestyle()
    Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_linewidth()
    Return the line width in points.

get_ls()
    Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_lw()
    Return the line width in points.

get_patch_transform()
    Return the Transform instance which takes patch coordinates to data coordinates.

    For example, one may define a patch of a circle which represents a radius of 5 by providing
    coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate)
    by 5.

get_path()
    Return the path of this patch

get_transform()
    Return the Transform applied to the Patch.
get_verts()
    Return a copy of the vertices used in this patch

    If the patch contains Bezier curves, the curves will be interpolated by line segments. To access
    the curves as curves, use get_path().

get_window_extent(renderer=None)

set_aa(aa)
    alias for set_antialiased

set_alpha(alpha)
    Set the alpha transparency of the patch.

    ACCEPTS: float or None

set_antialiased(aa)
    Set whether to use antialiased rendering

    ACCEPTS: [True | False] or None for default

set_capstyle(s)
    Set the patch capstyle

    ACCEPTS: ['butt' | 'round' | 'projecting']

set_color(c)
    Set both the edgecolor and the facecolor.

    ACCEPTS: matplotlib color spec

    See also:

        set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_ec(color)
    alias for set_edgecolor

set_edgecolor(color)
    Set the patch edge color

    ACCEPTS: mpl color spec, or None for default, or 'none' for no color

set_facecolor(color)
    Set the patch face color

    ACCEPTS: mpl color spec, or None for default, or 'none' for no color

set_fc(color)
    alias for set_facecolor

set_fill(b)
    Set whether to fill the patch

    ACCEPTS: [True | False]
**set_hatch(hatch)**

Set the hatching pattern

*hatch* can be one of:

```
/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
O  - large circle
.  - dots
*  - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

`ACCEPTS: [’/’ | ‘\’ | ‘|’ | ‘-’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]`

**set_joinstyle(s)**

Set the patch joinstyle

`ACCEPTS: [’miter’ | ‘round’ | ‘bevel’]`

**set_linestyle(ls)**

Set the patch linestyle

`ACCEPTS: [’solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’]`

**set_linewidth(w)**

Set the patch linewidth in points

`ACCEPTS: float or None for default`

**set_ls(ls)**

alias for `set_linestyle`

**set_lw(lw)**

alias for `set_linewidth`

**update_from(other)**

Updates this `Patch` from the properties of `other`.

**validCap = (u’butt’, u’round’, u’projecting’)**

**validJoin = (u’miter’, u’round’, u’bevel’)**

**zorder = 1**
class matplotlib.patches.PathPatch(path, **kwargs)

Bases: matplotlib.patches.Patch

A general polycurve path patch.

*path* is a matplotlib.path.Path object.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

Patch For additional kwargs

get_path()
Bases: `matplotlib.patches.Patch`

A general polygon patch.

`xy` is a numpy array with shape Nx2.

If `closed` is `True`, the polygon will be closed so the starting and ending points are the same.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>axes</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

**Patch** For additional kwargs

**get_closed()**

Returns if the polygon is closed

**Returns closed** : bool
If the path is closed

get_path()

Get the path of the polygon

**Returns path**: Path

The Path object for the polygon

get_xy()

Get the vertices of the path

**Returns vertices**: numpy array

The coordinates of the vertices as a Nx2 ndarray.

set_closed(closed)

Set if the polygon is closed

**Parameters closed**: bool

True if the polygon is closed

set_xy(xy)

Set the vertices of the polygon

**Parameters xy**: numpy array or iterable of pairs

The coordinates of the vertices as a Nx2 ndarray or iterable of pairs.

xy

Set/get the vertices of the polygon. This property is provided for backward compatibility with matplotlib 0.91.x only. New code should use `get_xy()` and `set_xy()` instead.

class matplotlib.patches.Rectangle(xy, width, height, angle=0.0, **kwargs)

**Bases**: matplotlib.patches.Patch

Draw a rectangle with lower left at \( xy = (x, y) \) with specified \( width \) and \( height \).

**angle**

rotation in degrees (anti-clockwise)

**fill** is a boolean indicating whether to fill the rectangle

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
</tbody>
</table>

Continued on next page
Table 60.12 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

`contains(mouseevent)`

`get_bbox()`

`get_height()`

  Return the height of the rectangle

`get_patch_transform()`

`get_path()`

  Return the vertices of the rectangle

`get_width()`

  Return the width of the rectangle

`get_x()`

  Return the left coord of the rectangle

`get_xy()`

  Return the left and bottom coords of the rectangle

`get_y()`

  Return the bottom coord of the rectangle

`set_bounds(*args)`

  Set the bounds of the rectangle: l,b,w,h

  ACCEPTS: (left, bottom, width, height)
set_height(h)
    Set the width rectangle
    ACCEPTS: float

set_width(w)
    Set the width rectangle
    ACCEPTS: float

set_x(x)
    Set the left coord of the rectangle
    ACCEPTS: float

set_xy(xy)
    Set the left and bottom coords of the rectangle
    ACCEPTS: 2-item sequence

set_y(y)
    Set the bottom coord of the rectangle
    ACCEPTS: float

xy
    Return the left and bottom coords of the rectangle

class matplotlib.patches.RegularPolygon(xy, numVertices, radius=5, orientation=0, **kwargs)

Bases: matplotlib.patches.Patch

A regular polygon patch.

Constructor arguments:
xy A length 2 tuple (x, y) of the center.
numVertices the number of vertices.
radius The distance from the center to each of the vertices.
orientation rotates the polygon (in radians).
Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
</tbody>
</table>

Continued on next page
Table 60.13 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_patch_transform()

get_path()

numvertices

orientation

radius

xy

class matplotlib.patches.Shadow(patch, ox, oy, props=None, **kwargs)

Bases: matplotlib.patches.Patch

Create a shadow of the given patch offset by ox, oy. props, if not None, is a patch property update dictionary. If None, the shadow will have have the same color as the face, but darkened.

kwargs are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/']</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
draw(renderer)

get_patch_transform()

get_path()
```

class matplotlib.patches.Wedge(center, r, theta1, theta2, width=None, **kwargs)
Bases: matplotlib.patches.Patch

Wedge shaped patch.

Draw a wedge centered at $x, y$ center with radius $r$ that sweeps $\theta_1$ to $\theta_2$ (in degrees). If $width$ is given, then a partial wedge is drawn from inner radius $r - width$ to outer radius $r$. 
Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[/</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**get_path()**

**set_center**(center)

**set_radius**(radius)

**set_theta1**(theta1)

**set_theta2**(theta2)
**set_width**(width)

```
class matplotlib.patches.YAArrow(figure, xytip, xybase, width=4, frac=0.1, headwidth=12, **kwargs)
```

Bases: matplotlib.patches.Patch

Yet another arrow class.

This is an arrow that is defined in display space and has a tip at \(x_1, y_1\) and a base at \(x_2, y_2\).

Constructor arguments:
- **xytip** \((x, y)\) location of arrow tip
- **xybase** \((x, y)\) location the arrow base mid point
- **figure** The Figure instance (fig.dpi)
- **width** The width of the arrow in points
- **frac** The fraction of the arrow length occupied by the head
- **headwidth** The width of the base of the arrow head in points

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
</tbody>
</table>

Continued on next page
Table 60.16 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```
get_patch_transform()

get_path()

getpoints(x1, y1, x2, y2, k)
For line segment defined by (x1, y1) and (x2, y2) return the points on the line that is perpendicular to the line and intersects (x2, y2) and the distance from (x2, y2) of the returned points is k.

matplotlib.patches.bbox_artist(artist, renderer, props=None, fill=True)
This is a debug function to draw a rectangle around the bounding box returned by get_window_extent() of an artist, to test whether the artist is returning the correct bbox.
props is a dict of rectangle props with the additional property ‘pad’ that sets the padding around the bbox in points.

matplotlib.patches.draw_bbox(bbox, renderer, color=’k’, trans=None)
This is a debug function to draw a rectangle around the bounding box returned by get_window_extent() of an artist, to test whether the artist is returning the correct bbox.
```
61.1 matplotlib.path

A module for dealing with the polylines used throughout matplotlib.

The primary class for polyline handling in matplotlib is `Path`. Almost all vector drawing makes use of Paths somewhere in the drawing pipeline.

Whilst a `Path` instance itself cannot be drawn, there exists `Artist` subclasses which can be used for convenient Path visualisation - the two most frequently used of these are `PathPatch` and `PathCollection`.

class matplotlib.path.Path(vertices, codes=None, _interpolation_steps=1, closed=False, read-only=False)

Bases: object

`Path` represents a series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:
- `vertices`: an Nx2 float array of vertices
- `codes`: an N-length uint8 array of vertex types

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices as well as three codes `CURVE3`.

The code types are:
- **STOP** [1 vertex (ignored)] A marker for the end of the entire path (currently not required and ignored)
- **MOVETO** [1 vertex] Pick up the pen and move to the given vertex.
- **LINETO** [1 vertex] Draw a line from the current position to the given vertex.
- **CURVE3** [1 control point, 1 endpoint] Draw a quadratic Bezier curve from the current position, with the given control point, to the given end point.
- **CURVE4** [2 control points, 1 endpoint] Draw a cubic Bezier curve from the current position, with the given control points, to the given end point.
- **CLOSEPOLY** [1 vertex (ignored)] Draw a line segment to the start point of the current polyline.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use `iter_segments()` or `cleaned()` to get the vertex/code pairs. This is important, since many `Path` objects, as an optimization, do not store a `codes` at all, but have a default one provided for them by `iter_segments()`.

**Note:** The vertices and codes arrays should be treated as immutable – there are a number of optimiza-
Create a new path with the given vertices and codes.

**Parameters**

- **vertices**: array_like
  
  The \((n, 2)\) float array, masked array or sequence of pairs representing the vertices of the path.

  If *vertices* contains masked values, they will be converted to NaNs which are then handled correctly by the Agg PathIterator and other consumers of path data, such as `iter_segments()`.

- **codes**: {None, array_like}, optional
  
  n-length array integers representing the codes of the path. If not None, codes must be the same length as vertices. If None, *vertices* will be treated as a series of line segments.

- **_interpolation_steps**: int, optional
  
  Used as a hint to certain projections, such as Polar, that this path should be linearly interpolated immediately before drawing. This attribute is primarily an implementation detail and is not intended for public use.

- **closed**: bool, optional
  
  If *codes* is None and closed is True, vertices will be treated as line segments of a closed polygon.

- **readonly**: bool, optional
  
  Makes the path behave in an immutable way and sets the vertices and codes as read-only arrays.

**CLOSEPOLY** = 79

**CURVE3** = 3

**CURVE4** = 4

**LINETO** = 2

**MOVETO** = 1

**NUM_VERTICES_FOR_CODE** = \{0: 1, 1: 1, 2: 1, 3: 2, 4: 3, 79: 1\}

A dictionary mapping Path codes to the number of vertices that the code expects.

**STOP** = 0

**classmethod** `arc(theta1, theta2, n=None, is_wedge=False)`

Return an arc on the unit circle from angle \(theta1\) to angle \(theta2\) (in degrees).

If \(n\) is provided, it is the number of spline segments to make. If \(n\) is not provided, the number of spline segments is determined based on the delta between \(theta1\) and \(theta2\).

Masionobe, L. 2003. *Drawing an elliptical arc using polylines, quadratic or cubic Bezier curves.*
class method circle(center=(0.0, 0.0), radius=1.0, readonly=False)
Return a Path representing a circle of a given radius and center.

Parameters center : pair of floats
    The center of the circle. Default (0, 0).
radius : float
    The radius of the circle. Default is 1.
readonly : bool
    Whether the created path should have the "readonly" argument
    set when creating the Path instance.

Notes

The circle is approximated using cubic Bezier curves. This uses 8 splines around the circle
using the approach presented here:
    Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

cleaned(transform=None, remove_nans=False, clip=None, quantize=False, simplify=False,
curves=False, stroke_width=1.0, snap=False, sketch=None)
Cleans up the path according to the parameters returning a new Path instance.

See also:
See iter_segments() for details of the keyword arguments.

Returns Path instance with cleaned up vertices and codes.

clip_to_bbox(bbox, inside=True)
Clip the path to the given bounding box.

The path must be made up of one or more closed polygons. This algorithm will not behave
correctly for unclosed paths.

If inside is True, clip to the inside of the box, otherwise to the outside of the box.

code_type
    alias of uint8

codes
    The list of codes in the Path as a 1-D numpy array. Each code is one of STOP, MOVETO,
    LINETO, CURVE3, CURVE4 or CLOSEPOLY. For codes that correspond to more than one vertex
    (CURVE3 and CURVE4), that code will be repeated so that the length of self.vertices and
    self.codes is always the same.
contains_path(path, transform=None)
Returns True if this path completely contains the given path.

If transform is not None, the path will be transformed before performing the test.
contains_point(point, transform=None, radius=0.0)
Returns True if the path contains the given point.

If transform is not None, the path will be transformed before performing the test.
radius allows the path to be made slightly larger or smaller.

contains_points(points, transform=None, radius=0.0)
Returns a bool array which is True if the path contains the corresponding point.
If transform is not None, the path will be transformed before performing the test.
radius allows the path to be made slightly larger or smaller.

copy()
Returns a shallow copy of the Path, which will share the vertices and codes with the source Path.

deepcopy()
Returns a deepcopy of the Path. The Path will not be readonly, even if the source Path is.

get_extents(transform=None)
Returns the extents (xmin, ymin, xmax, ymax) of the path.
Unlike computing the extents on the vertices alone, this algorithm will take into account the curves and deal with control points appropriately.

has_nonfinite
True if the vertices array has nonfinite values.

classmethod hatch(hatchpattern, density=6)
Given a hatch specifier, hatchpattern, generates a Path that can be used in a repeated hatching pattern. density is the number of lines per unit square.

interpolated(steps)
Returns a new path resampled to length N x steps. Does not currently handle interpolating curves.

intersects_bbox(bbox, filled=True)
Returns True if this path intersects a given Bbox.
filled, when True, treats the path as if it was filled. That is, if one path completely encloses the other, intersects_path() will return True.

intersects_path(other, filled=True)
Returns True if this path intersects another given path.
filled, when True, treats the paths as if they were filled. That is, if one path completely encloses the other, intersects_path() will return True.

iter_segments(transform=None, remove_nans=True, clip=None, snap=False, stroke_width=1.0, simplify=None, curves=True, sketch=None)
Iterates over all of the curve segments in the path. Each iteration returns a 2-tuple (vertices, code), where vertices is a sequence of 1 - 3 coordinate pairs, and code is one of the Path codes. Additionally, this method can provide a number of standard cleanups and conversions to the path.

Parameters transform : None or Transform instance
If not None, the given affine transformation will be applied to the path.
remove_nans : {False, True}, optional
If True, will remove all NaNs from the path and insert MOVETO commands to skip over them.

**clip** : None or sequence, optional
If not None, must be a four-tuple \((x1, y1, x2, y2)\) defining a rectangle in which to clip the path.

**snap** : None or bool, optional
If None, auto-snap to pixels, to reduce fuzziness of rectilinear lines. If True, force snapping, and if False, don’t snap.

**stroke_width** : float, optional
The width of the stroke being drawn. Needed as a hint for the snapping algorithm.

**simplify** : None or bool, optional
If True, perform simplification, to remove vertices that do not affect the appearance of the path. If False, perform no simplification. If None, use the should_simplify member variable.

**curves** : \{True, False\}, optional
If True, curve segments will be returned as curve segments. If False, all curves will be converted to line segments.

**sketch** : None or sequence, optional
If not None, must be a 3-tuple of the form (scale, length, randomness), representing the sketch parameters.

*classmethod* `make_compound_path(*args)`
Make a compound path from a list of Path objects.

*classmethod* `make_compound_path_from_polys(XY)`
Make a compound path object to draw a number of polygons with equal numbers of sides. \(XY\) is a \((\text{numpolys} \times \text{numsides} \times 2)\) numpy array of vertices. Return object is a `Path`
readonly
  True if the Path is read-only.

should_simplify
  True if the vertices array should be simplified.

simplify_threshold
  The fraction of a pixel difference below which vertices will be simplified out.

to_polygons(transform=None, width=0, height=0)
  Convert this path to a list of polygons. Each polygon is an Nx2 array of vertices. In other words, each polygon has no MOVETO instructions or curves. This is useful for displaying in backends that do not support compound paths or Bezier curves, such as GDK.

  If width and height are both non-zero then the lines will be simplified so that vertices outside of (0, 0), (width, height) will be clipped.

transformed(transform)
  Return a transformed copy of the path.

See also:
  matplotlib.transforms.TransformedPath A specialized path class that will cache the transformed result and automatically update when the transform changes.

classmethod unit_circle()
  Return the readonly Path of the unit circle.
For most cases, `Path.circle()` will be what you want.

**classmethod unit_circle_righthalf()**

Return a `Path` of the right half of a unit circle. The circle is approximated using cubic Bezier curves. This uses 4 splines around the circle using the approach presented here:

Lancaster, Don. *Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.*

**classmethod unit_rectangle()**

Return a `Path` instance of the unit rectangle from (0, 0) to (1, 1).

**classmethod unit_regular_asterisk(numVertices)**

Return a `Path` for a unit regular asterisk with the given numVertices and radius of 1.0, centered at (0, 0).

**classmethod unit_regular_polygon(numVertices)**

Return a `Path` instance for a unit regular polygon with the given numVertices and radius of 1.0, centered at (0, 0).

**classmethod unit_regular_star(numVertices, innerCircle=0.5)**

Return a `Path` for a unit regular star with the given numVertices and radius of 1.0, centered at (0, 0).

**vertices**

The list of vertices in the `Path` as an Nx2 numpy array.

**classmethod wedge(theta1, theta2, n=None)**

Return a wedge of the unit circle from angle theta1 to angle theta2 (in degrees).

If n is provided, it is the number of spline segments to make. If n is not provided, the number of spline segments is determined based on the delta between theta1 and theta2.

**matplotlib.path.cleanup_path(*args, **kwargs)**

Deprecated since version 1.3: The cleanup_path function was deprecated in version 1.3. Use `path.Path.cleaned` instead.

**cleanup_path(path, trans, remove_nans, clip, snap, simplify, curves, sketch_params)**

**matplotlib.path.clip_path_to_rect(*args, **kwargs)**

Deprecated since version 1.3: The clip_path_to_rect function was deprecated in version 1.3. Use `path.Path.clip_to_bbox` instead.

**clip_path_to_rect(path, bbox, inside)**

**matplotlib.path.convert_path_to_polygons(*args, **kwargs)**

Deprecated since version 1.3: The convert_path_to_polygons function was deprecated in version 1.3. Use `path.Path.to_polygons` instead.

**convert_path_to_polygons(path, trans, width, height)**

**matplotlib.path.get_path_collection_extents(master_transform, paths, transforms, offsets, offset_transform)**

Given a sequence of `Path` objects, `Transform` objects and offsets, as found in a `PathCollection`, returns the bounding box that encapsulates all of them.

`master_transform` is a global transformation to apply to all paths

```
61.1. matplotlib.path
```
paths is a sequence of Path instances.

transforms is a sequence of Affine2D instances.

offsets is a sequence of (x, y) offsets (or an Nx2 array)

offset_transform is a Affine2D to apply to the offsets before applying the offset to the path.

The way that paths, transforms and offsets are combined follows the same method as for collections. Each is iterated over independently, so if you have 3 paths, 2 transforms and 1 offset, their combinations are as follows:

(A, A, A), (B, B, A), (C, A, A)

matplotlib.path.get_path_extents(*args, **kwargs)

Deprecated since version 1.3: The get_path_extents function was deprecated in version 1.3. Use path.Path.get_extents instead.

get_path_extents(path, trans)

matplotlib.path.get_paths_extents(paths, transforms=[])

Given a sequence of Path objects and optional Transform objects, returns the bounding box that encapsulates all of them.

paths is a sequence of Path instances.

transforms is an optional sequence of Affine2D instances to apply to each path.

matplotlib.path.path_in_path(*args, **kwargs)

Deprecated since version 1.3: The path_in_path function was deprecated in version 1.3. Use path.Path.contains_path instead.

path_in_path(a, atrans, b, btrans)

matplotlib.path.path_intersects_path(*args, **kwargs)

Deprecated since version 1.3: The path_intersects_path function was deprecated in version 1.3. Use path.Path.intersects_path instead.

path_intersects_path(p1, p2)

matplotlib.path.point_in_path(*args, **kwargs)

Deprecated since version 1.3: The point_in_path function was deprecated in version 1.3. Use path.Path.contains_point instead.

point_in_path(x, y, path, trans)

matplotlib.path.point_in_path_collection(*args, **kwargs)

Deprecated since version 1.3: The point_in_path_collection function was deprecated in version 1.3. Use collection.Collection.contains instead.

point_in_path_collection(x, y, r, trans, paths, transforms, offsets, offsetTrans, filled)

matplotlib.path.points_in_path(*args, **kwargs)

Deprecated since version 1.3: The points_in_path function was deprecated in version 1.3. Use path.Path.contains_points instead.

points_in_path(points, path, trans)
62.1 matplotlib.patheffects

Defines classes for path effects. The path effects are supported in Text, Line2D and Patch.

```python
class matplotlib.patheffects.AbstractPathEffect(offset=(0.0, 0.0))
Bases: object

A base class for path effects.

Subclasses should override the draw_path method to add effect functionality.

Parameters offset: pair of floats
The offset to apply to the path, measured in points.

draw_path(renderer, gc, tpath, affine, rgbFace=None)
Derived should override this method. The arguments are the same as
matplotlib.backend_bases.RendererBase.draw_path() except the first argument
is a renderer.

get_proxy_renderer(renderer)
Return a PathEffectRenderer instance for this PathEffect.
```

```python
class matplotlib.patheffects.Normal(offset=(0.0, 0.0))
Bases: matplotlib.patheffects.AbstractPathEffect

The “identity” PathEffect.

The Normal PathEffect’s sole purpose is to draw the original artist with no special path effect.

Parameters offset: pair of floats
The offset to apply to the path, measured in points.
```

```python
class matplotlib.patheffects.PathEffectRenderer(path_effects, renderer)
Bases: matplotlib.backend_bases.RendererBase

Implements a Renderer which contains another renderer.

This proxy then intercepts draw calls, calling the appropriate AbstractPathEffect draw method.
```

**Note:** Not all methods have been overridden on this RendererBase subclass. It may be necessary to
add further methods to extend the PathEffects capabilities further.

Parameters path_effects: iterable of AbstractPathEffect
The path effects which this renderer represents.

**renderer**: `matplotlib.backend_bases.RendererBase` instance

**copy_with_path_effect**(path_effects)

**draw_markers**(gc, marker_path, marker_trans, path, *args, **kwargs)

**draw_path**(gc, tpath, affine, rgbFace=None)

**draw_path_collection**(gc, master_transform, paths, *args, **kwargs)

**points_to_pixels**(points)

class `matplotlib.patheffects.PathPatchEffect`(offset=(0, 0), **kwargs)

Bases: `matplotlib.patheffects.AbstractPathEffect`

Draws a `PathPatch` instance whose Path comes from the original PathEffect artist.

**Parameters**

offset : pair of floats
  The offset to apply to the path, in points.

**kwargs :
  All keyword arguments are passed through to the `PathPatch` constructor. The properties which cannot be overridden are “path”, “clip_box” “transform” and “clip_path”.

**draw_path**(renderer, gc, tpath, affine, rgbFace)

class `matplotlib.patheffects.SimpleLineShadow`(offset=(2, -2), shadow_color=u’k’, alpha=0.3, rho=0.3, **kwargs)

Bases: `matplotlib.patheffects.AbstractPathEffect`

A simple shadow via a line.

**Parameters**

offset : pair of floats
  The offset to apply to the path, in points.

shadow_color : color
  The shadow color. Default is black. A value of None takes the original artist’s color with a scale factor of rho.

alpha : float
  The alpha transparency of the created shadow patch. Default is 0.3.

rho : float
  A scale factor to apply to the rgbFace color if shadow_rgbFace is None. Default is 0.3.

**kwargs

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

**draw_path**(renderer, gc, tpath, affine, rgbFace)

Overides the standard draw_path to add the shadow offset and necessary color changes for the shadow.
class matplotlib.patheffects.SimplePatchShadow(  
    offset=(2, -2),  
    shadow_rgbFace=None,  
    alpha=None,  
    patch_alpha=None,  
    rho=0.3, offset_xy=None, **kwargs)

Bases: matplotlib.patheffects.AbstractPathEffect

A simple shadow via a filled patch.

Parameters:
- **offset**: pair of floats
  The offset of the shadow in points.
- **shadow_rgbFace**: color
  The shadow color.
- **alpha**: float
  The alpha transparency of the created shadow patch. Default is 0.3. http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html
- **rho**: float
  A scale factor to apply to the rgbFace color if shadow_rgbFace is not specified. Default is 0.3.

**kwargs
Extra keywords are stored and passed through to AbstractPathEffect._update_gc().

draw_path (renderer, gc, tpath, affine, rgbFace)
Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

class matplotlib.patheffects.Stroke(  
    offset=(0, 0), **kwargs)
Bases: matplotlib.patheffects.AbstractPathEffect

A line based PathEffect which re-draws a stroke.

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

draw_path (renderer, gc, tpath, affine, rgbFace)
draw the path with updated gc.

class matplotlib.patheffects.withSimplePatchShadow(  
    offset=(2, -2),  
    shadow_rgbFace=None,  
    alpha=None,  
    patch_alpha=None,  
    rho=0.3, offset_xy=None, **kwargs)

Bases: matplotlib.patheffects.SimplePatchShadow

Adds a simple SimplePatchShadow and then draws the original Artist to avoid needing to call Normal.

Parameters:
- **offset**: pair of floats
  The offset of the shadow in points.
- **shadow_rgbFace**: color
  The shadow color.
- **alpha**: float
  The alpha transparency of the created shadow patch. Default is 0.3. http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html
rho : float
    A scale factor to apply to the rgbFace color if shadow_rgbFace is not specified. Default is 0.3.

**kwargs
    Extra keywords are stored and passed through to AbstractPathEffect._update_gc().

draw_path(renderer, gc, tpath, affine, rgbFace)

class matplotlib.patheffects.withStroke(offset=(0, 0), **kwargs)
    Bases: matplotlib.patheffects.Stroke

    Adds a simple Stroke and then draws the original Artist to avoid needing to call Normal.
    The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

draw_path(renderer, gc, tpath, affine, rgbFace)
63.1 matplotlib.pyplot

Provides a MATLAB-like plotting framework.

pylab combines pyplot with numpy into a single namespace. This is convenient for interactive work, but for programming it is recommended that the namespaces be kept separate, e.g.:

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1);
y = np.sin(x)
plt.plot(x, y)

matplotlib.pyplot.acorr(x, hold=None, **kwargs)
```

Plot the autocorrelation of \( x \).

**Parameters**

- \( x \) : sequence of scalar
- \( hold \) : boolean, optional, default: True
- \( detrend \) : callable, optional, default: mlab.detrend_none
  - x is detrended by the detrend callable. Default is no normalization.
- \( normed \) : boolean, optional, default: True
  - if True, normalize the data by the autocorrelation at the 0-th lag.
- \( usevlines \) : boolean, optional, default: True
  - if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.
- \( maxlags \) : integer, optional, default: 10
  - number of lags to show. If None, will return all \( 2 \times \text{len}(x) - 1 \) lags.

**Returns**

- \( (\text{lags}, c, \text{line}, b) \) : where:
  - \( \text{lags} \) are a length \( 2 \times \text{maxlags} + 1 \) lag vector.
  - \( c \) is the \( 2 \times \text{maxlags} + 1 \) auto correlation vector
  - \( \text{line} \) is a Line2D instance returned by plot.
  - \( b \) is the x-axis.

**Other Parameters**

- **linestyle** : Line2D prop, optional, default: None
  - Only used if usevlines is False.
- **marker** : string, optional, default: ‘o’
Notes

The cross correlation is performed with `numpy.correlate()` with mode = 2.

Examples

`xcorr` is top graph, and `acorr` is bottom graph.

```
Additional kwargs: hold = [True|False] overrides default hold state
```

```
matplotlib.pyplot.angle_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None,
    sides='default', hold=None, **kwargs)
```

Plot the angle spectrum.

Call signature:

```
angle_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
    pad_to=None, sides='default', **kwargs)
```

Compute the angle spectrum (wrapped phase spectrum) of x. Data is padded to a length of `pad_to` and the windowing function `window` is applied to the signal.

`x`: 1-D array or sequence  Array or sequence containing the data

Keyword arguments:
**Fs:** scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**window:** callable or ndarray A function or a vector of length *NFFT*. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides:** [‘default’| ‘onesided’| ‘twosided’ ] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to:** integer
The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

**Fc:** integer The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and down-sampled to baseband.

Returns the tuple (*spectrum*, *freqs*, *line*):

**spectrum:** 1-D array The values for the angle spectrum in radians (real valued)

**freqs:** 1-D array The frequencies corresponding to the elements in *spectrum*

**line:** a Line2D instance The line created by this function

**kwargs:** control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True, False]</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True, False]</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True, False]</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default', 'steps', 'steps-pre', 'steps-mid', 'steps-post']</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full', 'left', 'right', 'bottom', 'top', 'none']</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
</tbody>
</table>
Table 63.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle or ls</td>
<td>['-', '--', '-.', ':', 'None', ' ' ]</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True, False]</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None, int, length-2 tuple of int, slice, list/array of int, float, length-2 tuple of float]</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
See also:

- `magnitude_spectrum()` and `angle_spectrum()` plots the magnitudes of the corresponding frequencies.
- `phase_spectrum()` and `phase_spectrum()` plots the unwrapped version of this function.
- `specgram()` and `specgram()` can plot the angle spectrum of segments within the signal in a colormap.

Additional kwarg: `hold` = [True|False] overrides default hold state

```
matplotlib.pyplot.annotate(*args, **kwargs)
```

Create an annotation: a piece of text referring to a data point.

**Parameters**

- `s` : string
  - label
- `xy` : (x, y)
  - position of element to annotate
- `xytext` : (x, y), optional, default: None
  - position of the label `s`
- `xycoords` : string, optional, default: “data”
  - string that indicates what type of coordinates `xy` is. Examples: “figure points”, “figure pixels”, “figure fraction”, “axes points”, .... See `matplotlib.text.Annotation` for more details.
- `textcoords` : string, optional
  - string that indicates what type of coordinates `text` is. Examples: “figure points”, “figure pixels”, “figure fraction”, “axes points”, .... See
**arrowprops**: matplotlib.lines.Line2D properties, optional

Dictionary of line properties for the arrow that connects the annotation to the point. If the dictionary has a key `arrowstyle`, a FancyArrowPatch instance is created and drawn. See matplotlib.text.Annotation for more details on valid options. Default is None.

**Returns a**: Annotation

**Notes**

`arrowprops`, if not `None`, is a dictionary of line properties (see `matplotlib.lines.Line2D`) for the arrow that connects annotation to the point.

If the dictionary has a key `arrowstyle`, a FancyArrowPatch instance is created with the given dictionary and is drawn. Otherwise, a YAArrow patch instance is created and drawn. Valid keys for YAArrow are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>oftentimes it is convenient to have the arrowtip and base a bit away from</td>
</tr>
<tr>
<td></td>
<td>the text and annotated point. If ( d ) is the distance between the text</td>
</tr>
<tr>
<td></td>
<td>and annotated point, shrink will shorten the arrow so the tip and base are</td>
</tr>
<tr>
<td></td>
<td>shrink percent of the distance ( d ) away from the endpoints. i.e., shrink=0.05 is 5%</td>
</tr>
</tbody>
</table>

Valid keys for FancyArrowPatch are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1</td>
</tr>
<tr>
<td>?</td>
<td>any key for matplotlib.patches.PathPatch</td>
</tr>
</tbody>
</table>

`xycoords` and `textcoords` are strings that indicate the coordinates of `xy` and `xytext`, and may be one of the following values:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'figure points'</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure pixels'</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure fraction'</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>'axes points'</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>'axes pixels'</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>'axes fraction'</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>'data'</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>'offset points'</td>
<td>Specify an offset (in points) from the $xy$ value</td>
</tr>
<tr>
<td>'polar'</td>
<td>you can specify $\theta$, $r$ for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native “data” coordinate system.</td>
</tr>
</tbody>
</table>

If a ‘points’ or ‘pixels’ option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. e.g.:

```python
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of Transform or Artist. See Annotating Axes for more details.

The `annotation_clip` attribute controls the visibility of the annotation when it goes outside the axes area. If True, the annotation will only be drawn when the $xy$ is inside the axes. If False, the annotation will always be drawn regardless of its position. The default is None, which behave as True only if `xycoords` is “data”.

Additional kwags are `Text` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>or fontname or name</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>['center'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>['center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Examples

```
import matplotlib.pyplot as plt

# Example 1
fig, ax = plt.subplots()
ax.plot(range(6), [0, 1, 2, 3, 4, 5], 'o-')
ax.annotate('arrowstyle', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='arc3,arc'))
ax.annotate('arc', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='arc'))
ax.annotate('angle', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='arc3'))
ax.annotate('angle', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='arc3', angle=90))
ax.annotate('angle3', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='arc3', angle3=90))
ax.annotate('angle', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='fancy'))
ax.annotate('fancy', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='fancy'))
ax.annotate('simple', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='simple'))
ax.annotate('wedge', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='wedge'))
ax.annotate('wedge', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='wedge', head_width=5))
ax.annotate('wedge', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='wedge', head_width=5, head_length=5))
ax.annotate('wedge', xy=(1, 3), xytext=(4, 0), arrowprops=dict(arrowstyle='wedge', head_width=5, head_length=5, tail_width=5))
```

```
plt.show()
```
Add an arrow to the axes.

Call signature:

```python
arrow(x, y, dx, dy, hold=None, **kwargs)
```

Draws arrow on specified axis from \((x, y)\) to \((x + dx, y + dy)\). Uses FancyArrow patch to construct the arrow.

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use `annotate()` for example:

```python
tax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
             arrowprops=dict(arrowstyle="->"))
```

Optional kwargs control the arrow construction and properties:

**Constructor arguments**

- `width`: float (default: 0.001) width of full arrow tail
- `length_includes_head`: [True | False] (default: False) True if head is to be counted in calculating the length.
- `head_width`: float or None (default: 3*width) total width of the full arrow head
- `head_length`: float or None (default: 1.5 * head_width) length of arrow head
- `shape`: ["full", ‘left’, ‘right’] (default: ‘full’) draw the left-half, right-half, or full arrow
- `overhang`: float (default: 0) fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.
- `head_starts_at_zero`: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from `Patch`) are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt’</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 63.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>joinstyle</code></td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>['solid', 'dashed', 'dashdot', 'dotted']</td>
</tr>
<tr>
<td><code>linewidth</code></td>
<td>float or None for default</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True, False]</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True, False, None]</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>Transform instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True, False]</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**Example:**

![Example Diagram](image)

Additional kwargs: `hold` = [True|False] overrides default hold state

```python
matplotlib.pyplot.autoscale(enable=True, axis='both', tight=None)
```

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.
**enable:** [True | False | None] True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis:** ['x' | 'y' | 'both'] which axis to operate on; default is 'both'

**tight:** [True | False | None] If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat tight as False. The tight setting is retained for future autoscaling until it is explicitly changed.

Returns None.

`matplotlib.pyplot.autumn()`

set the default colormap to autumn and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.axes(*args, **kwargs)`

Add an axes to the figure.

The axes is added at position `rect` specified by:
- `axes()` by itself creates a default full `subplot(111)` window axis.
- `axes(rect, axisbg='w')` where `rect` = [left, bottom, width, height] in normalized (0, 1) units. `axisbg` is the background color for the axis, default white.
- `axes(h)` where `h` is an axes instance makes `h` the current axis. An Axes instance is returned.

<table>
<thead>
<tr>
<th>kwarg</th>
<th>Accepts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axisbg</td>
<td>color</td>
<td>the axes background color</td>
</tr>
<tr>
<td>frameon</td>
<td>[True</td>
<td>False]</td>
</tr>
<tr>
<td>sharex</td>
<td>otherax</td>
<td>current axes shares xaxis attribute with otherax</td>
</tr>
<tr>
<td>sharey</td>
<td>otherax</td>
<td>current axes shares yaxis attribute with otherax</td>
</tr>
<tr>
<td>polar</td>
<td>[True</td>
<td>False]</td>
</tr>
<tr>
<td>aspect</td>
<td>[str</td>
<td>num]</td>
</tr>
</tbody>
</table>

Examples:
- `examples/pylab_examples/axes_demo.py` places custom axes.
- `examples/pylab_examples/shared_axis_demo.py` uses `sharex` and `sharey`.

`matplotlib.pyplot.axhline(y=0, xmin=0, xmax=1, hold=|None, **kwargs)`

Add a horizontal line across the axis.

**Parameters**
- `y` : scalar, optional, default: 0
  y position in data coordinates of the horizontal line.
- `xmin` : scalar, optional, default: 0
  Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.
- `xmax` : scalar, optional, default: 1
  Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**Returns** `Line2D`

**See also:**
- `axhspan` for example plot and source code

**Additional**
Notes

kwargs are the same as kwargs to plot, and can be used to control the line properties. e.g.,

Examples

• draw a thick red hline at ‘y’ = 0 that spans the xrange:

```python
>>> axhline(linewidth=4, color='r')
```

• draw a default hline at ‘y’ = 1 that spans the xrange:

```python
>>> axhline(y=1)
```

• draw a default hline at ‘y’ = .5 that spans the the middle half of the xrange:

```python
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Valid kwargs are Line2D properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.pyplot.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[’-‘</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
</tbody>
</table>
Table 63.4 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```
matplotlib.pyplot.axhspan(ymin, ymax, xmin=0, xmax=1, hold=None, **kwargs)
```

Add a horizontal span (rectangle) across the axis.

Call signature:

```
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

* y coords are in data units and *x* coords are in axes (relative 0-1) units.

* Draw a horizontal span (rectangle) from *ymin* to *ymax*. With the default values of *xmin* = 0 and *xmax* = 1, this always spans the xrange, regardless of the xlim settings, even if you change them, e.g., with the `set_xlim()` command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the *y* location is in data coordinates.

* Return value is a `matplotlib.patches.Polygon` instance.

Examples:

* draw a gray rectangle from *y* = 0.25-0.75 that spans the horizontal extent of the axes:

````
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
```

Valid kwargs are `Polygon` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>or aa</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>axes</td>
<td></td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.5 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Additional kwargs: hold = [True]False] overrides default hold state

```
matplotlib.pyplot.axis(*v, **kwargs)
```

Convenience method to get or set axis properties.

Calling with no arguments:

```python
>>> axis()
```

returns the current axes limits \([x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}]\).

```python
>>> axis(v)
```

sets the min and max of the x and y axes, with \(v = [x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}]\).

```python
>>> axis('off')
```

turns off the axis lines and labels.

```python
>>> axis('equal')
```

changes limits of \(x\) or \(y\) axis so that equal increments of \(x\) and \(y\) have the same length; a circle is circular.
>>> axis('scaled')

achieves the same result by changing the dimensions of the plot box instead of the axis data limits.:  

>>> axis('tight')

changes x and y axis limits such that all data is shown. If all data is already shown, it will move it to the center of the figure without modifying (xmax - xmin) or (ymax - ymin). Note this is slightly different than in MATLAB.:  

>>> axis('image')

is ‘scaled’ with the axis limits equal to the data limits.:  

>>> axis('auto')

and:

>>> axis('normal')

are deprecated. They restore default behavior; axis limits are automatically scaled to make the data fit comfortably within the plot box.

if len(*v)==0, you can pass in xmin, xmax, ymin, ymax as kwargs selectively to alter just those limits without changing the others.

The xmin, xmax, ymin, ymax tuple is returned 

See also:

xlim(), ylim() For setting the x- and y-limits individually.

matplotlib.pyplot.axvline(x=0, ymin=0, ymax=1, hold=None, **kwargs)

Add a vertical line across the axes.

Parameters x : scalar, optional, default: 0
    x position in data coordinates of the vertical line.

ymin : scalar, optional, default: 0
    Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

ymax : scalar, optional, default: 1
    Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

Returns Line2D

See also:

axhspan for example plot and source code

Additional
Examples

- draw a thick red vline at $x = 0$ that spans the yrange:

  ```python
g>>> axvline(linewidth=4, color='r')
  ```

- draw a default vline at $x = 1$ that spans the yrange:

  ```python
g>>> axvline(x=1)
  ```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

  ```python
g>>> axvline(x=.5, ymin=0.25, ymax=0.75)
  ```

Valid kwargs are `Line2D` properties, with the exception of `transform`:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with %%s conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgedwidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
</tbody>
</table>
Table 63.6 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

matplotlib.pyplot.axvspan(xmin, xmax, ymin=0, ymax=1, hold=None, **kwargs)

Add a vertical span (rectangle) across the axes.

Call signature:

axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)

x coords are in data units and y coords are in axes (relative 0-1) units.

Draw a vertical span (rectangle) from $x_{min}$ to $x_{max}$. With the default values of $y_{min} = 0$ and $y_{max} = 1$, this always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the set_ylim() command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the y location is in data coordinates.

Return value is the matplotlib.patches.Polygon instance.

Examples:

• draw a vertical green translucent rectangle from x=1.25 to 1.55 that spans the yrange of the axes:

  >>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)

Valid kwarg are Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased  or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
</tbody>
</table>
Table 63.7 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

axhspan() for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.bar(left, height, width=0.8, bottom=None, hold=None, **kwargs)

Make a bar plot.

Make a bar plot with rectangles bounded by:

left, left + width, bottom, bottom + height (left, right, bottom and top edges)

Parameters left : sequence of scalars

the x coordinates of the left sides of the bars

height : sequence of scalars

the heights of the bars

width : scalar or array-like, optional, default: 0.8

the width(s) of the bars

bottom : scalar or array-like, optional, default: None

the y coordinate(s) of the bars

color : scalar or array-like, optional

the colors of the bar faces

edgecolor : scalar or array-like, optional

the colors of the bar edges
**linewidth** : scalar or array-like, optional, default: None
width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.

**xerr** : scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**yerr** : scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**ecolor** : scalar or array-like, optional, default: None
specifies the color of errorbar(s)

**capsize** : integer, optional, default: 3
determines the length in points of the error bar caps

**error_kw** :
dictionary of kwags to be passed to errorbar method. *ecolor* and *capsize* may be specified here rather than as independent kwags.

**align** : ['edge' | 'center'], optional, default: 'edge'
If edge, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If center, interpret the left argument as the coordinates of the centers of the bars.

**orientation** : ‘vertical’ | ‘horizontal’, optional, default: ‘vertical’
The orientation of the bars.

**log** : boolean, optional, default: False
If true, sets the axis to be log scale

Returns `matplotlib.patches.Rectangle` instances.

See also:

**barh** Plot a horizontal bar plot.

### Notes

The optional arguments *color*, *edgecolor*, *linewidth*, *xerr*, and *yerr* can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: *xerr* and *yerr* are passed directly to `errorbar()`, so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwags:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td><code>matplotlib</code> color spec</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.8 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Examples

**Example:** A stacked bar chart.
Scores by group and gender

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.barbs(*args, **kw)
```
Plot a 2-D field of barbs.

Call signatures:

- `barb(U, V, **kw)`
- `barb(U, V, C, **kw)`
- `barb(X, Y, U, V, **kw)`
- `barb(X, Y, U, V, C, **kw)`

Arguments:

- `X, Y`: The x and y coordinates of the barb locations (default is head of barb; see `pivot` kwarg)
- `U, V`: Give the x and y components of the barb shaft
- `C`: An optional array used to map colors to the barbs

All arguments may be 1-D or 2-D arrays or sequences. If `X` and `Y` are absent, they will be generated as a uniform grid. If `U` and `V` are 2-D arrays but `X` and `Y` are 1-D, and if `len(X)` and `len(Y)` match the column and row dimensions of `U`, then `X` and `Y` will be expanded with `numpy.meshgrid()`.

`U, V, C` may be masked arrays, but masked `X, Y` are not supported at present.

Keyword arguments:
**length**: Length of the barb in points; the other parts of the barb are scaled against this. Default is 9

**pivot**: [‘tip’ | ‘middle’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot. Default is ‘tip’

**barbcolor**: [color | color sequence] Specifies the color all parts of the barb except any flags. This parameter is analogous to the `edgecolor` parameter for polygons, which can be used instead. However this parameter will override facecolor.

**flagcolor**: [color | color sequence] Specifies the color of any flags on the barb. This parameter is analogous to the `facecolor` parameter for polygons, which can be used instead. However this parameter will override facecolor. If this is not set (and `C` has not either) then `flagcolor` will be set to match `barbcolor` so that the barb has a uniform color. If `C` has been set, `flagcolor` has no effect.

**sizes**: A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- ‘spacing’ - space between features (flags, full/half bars)
- ‘height’ - height (distance from shaft to top) of a flag or full barb
- ‘width’ - width of a flag, twice the width of a full barb
- ‘emptybarb’ - radius of the circle used for low magnitudes

**fill_empty**: A flag on whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False

**rounding**: A flag to indicate whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple. Default is True

**barb_increments**: A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- ‘half’ - half barbs (Default is 5)
- ‘full’ - full barbs (Default is 10)
- ‘flag’ - flags (default is 50)

**flip_barb**: Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for each individual barb. Normal behavior is for the bars and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere.) Default is False

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:
The largest increment is given by a triangle (or “flag”). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

lineweights and edgecolors can be used to customize the barb. Additional PolyCollection keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Example:
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.barh(bottom, width, height=0.8, left=None, hold=None, **kwargs)
```

Make a horizontal bar plot.

Make a horizontal bar plot with rectangles bounded by:
- `left`, `left + width`, `bottom`, `bottom + height` (left, right, bottom and top edges)
- `bottom`, `width`, `height`, and `left` can be either scalars or sequences

**Parameters**
- `bottom` : scalar or array-like
  - the y coordinate(s) of the bars
- `width` : scalar or array-like
  - the width(s) of the bars
- `height` : sequence of scalars, optional, default: 0.8
  - the heights of the bars
- `left` : sequence of scalars
  - the x coordinates of the left sides of the bars

**Returns**
- `matplotlib.patches.Rectangle` instances.

**Other Parameters**
- `color` : scalar or array-like, optional
  - the colors of the bars
- `edgecolor` : scalar or array-like, optional
  - the colors of the bar edges
- `linewidth` : scalar or array-like, optional, default: None
  - width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.
**xerr**: scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**yerr**: scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**ecolor**: scalar or array-like, optional, default: None
specifies the color of errorbar(s)

**capsize**: integer, optional, default: 3
determines the length in points of the error bar caps

**error_kw**: dictionary of kwargs to be passed to errorbar method. `ecolor` and `capsize` may be specified here rather than as independent kwargs.

**align**: 
If `edge`, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If `center`, interpret the left argument as the coordinates of the centers of the bars.

**orientation**: ‘vertical’ | ‘horizontal’, optional, default: ‘vertical’
The orientation of the bars.

**log**: boolean, optional, default: False
If true, sets the axis to be log scale

See also:

**bar** Plot a vertical bar plot.

Additional

Notes

The optional arguments `color`, `edgecolor`, `linewidth`, `xerr`, and `yerr` can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: `xerr` and `yerr` are passed directly to `errorbar()`, so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased or aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor or ec</code></td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td><code>facecolor or fc</code></td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**matplotlib.pyplot.bone()**

set the default colormap to bone and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.box(on=None)**

Turn the axes box on or off. on may be a boolean or a string, ‘on’ or ‘off’.

If on is None, toggle state.

**matplotlib.pyplot.boxplot(x, notch=False, sym=None, vert=True, whis=1.5, positions=None, widths=None, patch_artist=False, bootstrap=None, usermedians=None, conf_intervals=None, meanline=False, showmeans=False, showcaps=True, showbox=True, showfliers=True, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whiskerprops=None, manage_xticks=True, hold=None)**

Make a box and whisker plot.

Call signature:

```python
boxplot(self, x, notch=False, sym='b+', vert=True, whis=1.5, positions=None, widths=None, patch_artist=False, bootstrap=None, usermedians=None, conf_intervals=None, meanline=False, showmeans=False, showcaps=True, showbox=True, showfliers=True, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whiskerprops=None, manage_xticks=True):
```
Make a box and whisker plot for each column of $x$ or each vector in sequence $x$. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

**Parameters**

- **$x$**: Array or a sequence of vectors.
  - The input data.

- **notch** [bool, default = False] If False, produces a rectangular box plot. If True, will produce a notched box plot.

- **sym** [str or None, default = None] The default symbol for flier points.
  - Enter an empty string ("") if you don’t want to show fliers. If None, then the fliers default to ‘b+’ If you want more control use the flierprops kwarg.

- **vert** [bool, default = True] If True (default), makes the boxes vertical. If False, makes horizontal boxes.

- **whis** [float, sequence (default = 1.5) or string] As a float, determines the reach of the whiskers past the first and third quartiles (e.g., $Q_3 + \text{whis}\times\text{IQR}, \text{IQR} = \text{interquartile range}, Q_3-Q_1$). Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, whis can be the string ‘range’ to force the whiskers to the min and max of the data. In the edge case that the 25th and 75th percentiles are equivalent, whis will be automatically set to ‘range’.

- **bootstrap** [None (default) or integer] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap=None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine it’s 95% confidence intervals. Values between 1000 and 10000 are recommended.

- **usermedians** [array-like or None (default)] An array or sequence whose first dimension (or length) is compatible with $x$. This overrides the medians computed by matplotlib for each element of usermedians that is not None. When an element of usermedians == None, the median will be computed by matplotlib as normal.

- **conf_intervals** [array-like or None (default)] Array or sequence whose first dimension (or length) is compatible with $x$ and whose second dimension is 2. When the current element of conf_intervals is not None, the notch locations computed by matplotlib are overridden (assuming notch is True). When an element of conf_intervals is None, boxplot compute notches the method specified by the other kwargs (e.g., bootstrap).

- **positions** [array-like, default = [1, 2, ..., n]] Sets the positions of the
boxes. The ticks and limits are automatically set to match the positions.

widths [array-like, default = 0.5] Either a scalar or a vector and sets
the width of each box. The default is 0.5, or 0.15*(distance
between extreme positions) if that is smaller.

labels [sequence or None (default)] Labels for each dataset. Length
must be compatible with dimensions of x

patch_artist [bool, default = False] If False produces boxes with the
Line2D artist If True produces boxes with the Patch artist

showmeans [bool, default = False] If True, will toggle one the render-
ing of the means

showcaps [bool, default = True] If True, will toggle one the rendering
of the caps

showbox [bool, default = True] If True, will toggle one the rendering
of box

showfliers [bool, default = True] If True, will toggle one the rendering
of the fliers

boxprops [dict or None (default)] If provided, will set the plotting
style of the boxes

whiskerprops [dict or None (default)] If provided, will set the plotting
style of the whiskers

capprops [dict or None (default)] If provided, will set the plotting
style of the caps

flierprops [dict or None (default)] If provided, will set the plotting
style of the fliers

medianprops [dict or None (default)] If provided, will set the plotting
style of the medians

meanprops [dict or None (default)] If provided, will set the plotting
style of the means

meanline [bool, default = False] If True (and showmeans is True), will
try to render the mean as a line spanning the full width of the
box according to meanprops. Not recommended if shownotches
is also True. Otherwise, means will be shown as points.

Returns result: dict

A dictionary mapping each component of the boxplot to a list of the
matplotlib.pyplot.Line2D instances created. That dictionary has
the following keys (assuming vertical boxplots):

•boxes: the main body of the boxplot showing the quartiles and
the median’s confidence intervals if enabled.
•medians: horizontal lines at the median of each box.
•whiskers: the vertical lines extending to the most extreme, n-
outlier data points.
•caps: the horizontal lines at the ends of the whiskers.
•fliers: points representing data that extend beyond the whiskers
(outliers).
•means: points or lines representing the means.
Examples

Default

showmeans=True

meanline=True

Tufte Style
(showbox=False,
showcaps=False)

notch=True,
bootstrap=10000

showfliers=False
I never said they’d be pretty

Custom boxprops

Custom medianprops

and flierprops

whis="range"

Custom mean

as point

Custom mean

as line

whis=[15, 85]

#percentiles

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.broken_barh(xranges, yrange, hold=None, **kwargs)

Plot horizontal bars.

Call signature:

broken_barh(self, xranges, yrange, **kwargs)

A collection of horizontal bars spanning yrange with a sequence of xranges.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xranges</td>
<td>sequence of (xmin, xwidth)</td>
</tr>
<tr>
<td>yrange</td>
<td>sequence of (ymin, ywidth)</td>
</tr>
</tbody>
</table>

kwargs are matplotlib.collections.BrokenBarHCollection properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ '/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%%s' conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

these can either be a single argument, i.e.,:

```python
facecolors = 'black'
```

or a sequence of arguments for the various bars, i.e.,:

```python
facecolors = ('black', 'red', 'green')
```
Example:

```
0   50   100  150  200
seconds since start
Bill
Jim
race interrupted
```

Additional kwargs: `hold = [True|False]` overrides default hold state

```python
clabel(cs, **kwargs)
```

Adds labels to line contours in `cs`, where `cs` is a `ContourSet` object returned by `contour`.

```python
clabel(cs, v, **kwargs)
```

only labels contours listed in `v`.

Optional keyword arguments:

- `fontsize`: size in points or relative size e.g., 'smaller', 'x-large'
- `colors`:
  - if `None`, the color of each label matches the color of the corresponding contour
• if one string color, e.g., `colors = 'r'` or `colors = 'red'`, all labels will be plotted in this color
• if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

`inline`: controls whether the underlying contour is removed or not. Default is `True`.

`inline_spacing`: space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

`fmt`: a format string for the label. Default is ‘%1.3f’ Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., `fmt[level]=string`), or it can be any callable, such as a `Formatter` instance, that returns a string when called with a numeric contour level.

`manual`: if `True`, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location). `manual` can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.

`rightside_up`: if `True` (default), label rotations will always be plus or minus 90 degrees from level.

`use_clabeltext`: if `True` (default is False), `ClabelText` class (instead of `matplotlib.Text`) is used to create labels. `ClabelText` recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.clf()
```
Clear the current figure.

```python
matplotlib.pyplot.clim(vmin=None, vmax=None)
```
Set the color limits of the current image.

To apply clim to all axes images do:

```python
clim(0, 0.5)
```
If either vmin or vmax is None, the image min/max respectively will be used for color scaling.

If you want to set the clim of multiple images, use, for example:

```python
for im in gca().get_images():
    im.set_clim(0, 0.05)
```

```python
matplotlib.pyplot.close(*args)
```
Close a figure window.

```python
close() by itself closes the current figure
```

```python
close(h) where h is a Figure instance, closes that figure
```

```python
close(num) closes figure number num
```
close(name) where name is a string, closes figure with that label

close('all') closes all the figure windows

matplotlib.pyplot.cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x7f8bf872f320>, window=<function window_hanning at 0x7f8bf872f050>, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, hold=None, **kwargs)

Plot the coherence between x and y.

Call signature:

cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none, window = mlab.window_hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, **kwargs)

Plot the coherence between x and y. Coherence is the normalized cross spectral density:

\[ C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \]  

63.1.

Keyword arguments:

Fs: scalar  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

window: callable or ndarray  A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides: [ 'default' | 'onesided' | 'twosided' ]  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

pad_to: integer  The number of points to which the data segment is padded when performing the FFT. This can be different from NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to NFFT

NFFT: integer  The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

detrend: [ ‘default’ | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’ ] or callable  The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().
**scale_by_freq**: boolean

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz\(^{-1}\). This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**: integer

The number of points of overlap between blocks. The default value is 0 (no overlap).

**Fc**: integer

The center frequency of \(x\) (defaults to 0), which offsets the \(x\) extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and down-sampled to baseband.

The return value is a tuple \((Cxy, f)\), where \(f\) are the frequencies of the coherence vector.

kwarg values are applied to the lines.

References:


kwarg values control the **Line2D** properties of the coherence plot:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <strong>Axes</strong> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <strong>matplotlib.transforms.Bbox</strong> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(<strong>Path</strong>,<strong>Transform</strong>)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <strong>matplotlib.figure.Figure</strong> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid <strong>marker</strong> style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
</tbody>
</table>
Table 63.12 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

```
Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.colorbar(mappable=None, cax=None, ax=None, **kw)
```
Add a colorbar to a plot.

Function signatures for the pyplot interface; all but the first are also method signatures for the colorbar() method:

colorbar(**kwargs)

colorbar(mappable, **kwargs)

colorbar(mappable, cax=cax, **kwargs)

colorbar(mappable, ax=ax, **kwargs)

arguments:

mappable the Image, ContourSet, etc. to which the colorbar applies; this argument is mandatory for the colorbar() method but optional for the colorbar() function, which sets the default to the current image.

keyword arguments:

cax None | axes object into which the colorbar will be drawn

ax None | parent axes object(s) from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.

use gridspec False | If cax is None, a new cax is created as an instance of Axes. If ax is an instance of Subplot and use_gridspec is True, cax is created as an instance of Subplot using the grid_spec module.

Additional keyword arguments are of two kinds:

axes properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>pananchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

colorbar properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extend</td>
<td>[ 'neither'</td>
</tr>
<tr>
<td>extend_frac</td>
<td>[ None</td>
</tr>
<tr>
<td>extend_direct</td>
<td>[ False</td>
</tr>
<tr>
<td>spacing</td>
<td>[ 'uniform'</td>
</tr>
<tr>
<td>ticks</td>
<td>[ None</td>
</tr>
<tr>
<td>format</td>
<td>[ None</td>
</tr>
<tr>
<td>drawedges</td>
<td>False</td>
</tr>
</tbody>
</table>

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundaries</td>
<td>None or a sequence</td>
</tr>
<tr>
<td>values</td>
<td>None or a sequence which must be of length 1 less than the sequence of boundaries. For each region delimited by adjacent entries in boundaries, the color mapped to the corresponding value in values will be used.</td>
</tr>
</tbody>
</table>

If mappable is a ContourSet, its extend kwarg is included automatically.

Note that the shrink kwarg provides a simple way to keep a vertical colorbar, for example, from being taller than the axes of the mappable to which the colorbar is attached; but it is a manual method requiring some trial and error. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of shrink.
For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewer (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers not matplotlib. As a workaround the colorbar can be rendered with overlapping segments:

cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()

However this has negative consequences in other circumstances. Particularly with semi transparent images (alpha < 1) and colorbar extensions and is not enabled by default see (issue #1188).

**returns:** Colorbar instance; see also its base class, ColorbarBase. Call the set_label() method to label the colorbar.

```python
matplotlib.pyplot.colors()
```

This is a do-nothing function to provide you with help on how matplotlib handles colors.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

<table>
<thead>
<tr>
<th>Alias</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>'w'</td>
<td>white</td>
</tr>
</tbody>
</table>

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```python
color = '#eefeff'
```

or you can pass an R,G,B tuple, where each of R,G,B are in the range [0,1].

You can also use any legal html name for a color, for example:

```python
color = 'red'
color = 'burlywood'
color = 'chartreuse'
```

The example below creates a subplot with a dark slate gray background:

```python
subplot(111, axisbg=(0.1843, 0.3098, 0.3098))
```

Here is an example that creates a pale turquoise title:

```python
title('Is this the best color?', color='afeeee')
```
Connect event with string \texttt{s} to \texttt{func}. The signature of \texttt{func} is:

\begin{verbatim}
def func(event)

where event is a \texttt{matplotlib.backend_bases.Event}. The following events are recognized
\begin{itemize}
\item 'button_press_event'
\item 'button_release_event'
\item 'draw_event'
\item 'key_press_event'
\item 'key_release_event'
\item 'motion_notify_event'
\item 'pick_event'
\item 'resize_event'
\item 'scroll_event'
\item 'figure_enter_event',
\item 'figure_leave_event',
\item 'axes_enter_event',
\item 'axes_leave_event'
\item 'close_event'
\end{itemize}

For the location events (button and key press/release), if the mouse is over the axes, the variable \texttt{event.inaxes} will be set to the \texttt{Axes} the event occurs is over, and additionally, the variables \texttt{event.xdata} and \texttt{event.ydata} will be defined. This is the mouse location in data coords. See \texttt{KeyEvent} and \texttt{MouseEvent} for more info.

Return value is a connection id that can be used with \texttt{mpl_disconnect()}

Example usage:

\begin{verbatim}
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
\end{verbatim}

\texttt{matplotlib.pyplot.contour(*args, **kwargs)}

Plot contours.

\texttt{contour()} and \texttt{contourf()} draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

\texttt{contourf()} differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to \texttt{contour()}.

Call signatures:

\begin{verbatim}
contour(Z)

make a contour plot of an array \texttt{Z}. The level values are chosen automatically.

contour(X,Y,Z)
\end{verbatim}
$X$, $Y$ specify the (x, y) coordinates of the surface

\begin{verbatim}
contour(Z,N)
contour(X,Y,Z,N)
\end{verbatim}

contour $N$ automatically-chosen levels.

\begin{verbatim}
contour(Z,V)
contour(X,Y,Z,V)
\end{verbatim}

draw contour lines at the values specified in sequence $V$

\begin{verbatim}
contourf(..., V)
\end{verbatim}

fill the $\text{len}(V)-1$ regions between the values in $V$

\begin{verbatim}
contour(Z, **kwargs)
\end{verbatim}

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

$X$ and $Y$ must both be 2-D with the same shape as $Z$, or they must both be 1-D such that $\text{len}(X)$ is the number of columns in $Z$ and $\text{len}(Y)$ is the number of rows in $Z$.

\begin{verbatim}
C = contour(...) returns a QuadContourSet object.
\end{verbatim}

Optional keyword arguments:

\begin{verbatim}
colors: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

alpha: float The alpha blending value

cmap: [ None | Colormap ] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

norm: [ None | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

vmin, vmax: [ None | scalar ] If not None, either or both of these values will be supplied to the matplotlib.colors.Normalize instance, overriding the default color scaling based on levels.

levels: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass levels=[0]

origin: [ None | ‘upper’ | ‘lower’ | ‘image’ ] If None, the first value of $Z$ will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if $X$ and $Y$ are specified in the call to contour.

extent: [ None | (x0,x1,y0,y1) ]
\end{verbatim}
If `origin` is not `None`, then `extent` is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If `origin` is `None`, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1].

This keyword is not active if X and Y are specified in the call to `contour`.

**locator**: `[None | ticker.Locator subclass]` If `locator` is `None`, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

**extend**: `[‘neither’ | ‘both’ | ‘min’ | ‘max’]` Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

**xunits, yunits**: `[None | registered units]` Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

**antialiased**: `[True | False]` enable antialiasing, overriding the defaults. For filled contours, the default is `True`. For line contours, it is taken from `rcParams[‘lines.antialiased’]`.

**linewidths**: `[None | number | tuple of numbers]` If `linewidths` is `None`, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles**: `[None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’]` If `linestyles` is `None`, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc contour.negative_linestyle` setting.

`linestyles` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**ncol**: `[0 | integer]` If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly `ncol` by `ncol` points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless `antialiased` is `False`.

**hatches**: A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: `contourf` fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:
$z_1 < z \leq z_2$

There is one exception: if the lowest boundary coincides with the minimum value of the $z$ array, then that minimum value will be included in the lowest interval.

Examples:
labels at selected locations
Single color - negative contours solid

matplotlib.pyplot
63.1. matplotlib.pyplot
1431
Listed colors (3 masked regions)
Additional kwargs: hold = [True,False] overrides default hold state

**matplotlib.pyplot**.contourf(*args, **kwargs)
Plot contours.

`contour()` and `contourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

`contourf()` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `contour()`.

Call signatures:

`contour(Z)`
make a contour plot of an array Z. The level values are chosen automatically.

`contour(X,Y,Z)`

X, Y specify the (x, y) coordinates of the surface

`contour(Z,N)`
`contour(X,Y,Z,N)`

contour N automatically-chosen levels.
contour(Z, V)
ccontour(X, Y, Z, V)

draw contour lines at the values specified in sequence V

ccontourf(..., V)
fill the \( \text{len}(V) - 1 \) regions between the values in \( V \)

contour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

\( X \) and \( Y \) must both be 2-D with the same shape as \( Z \), or they must both be 1-D such that \( \text{len}(X) \) is the number of columns in \( Z \) and \( \text{len}(Y) \) is the number of rows in \( Z \).

\( C = \text{contour}(...) \) returns a QuadContourSet object.

Optional keyword arguments:

\textbf{colors}: [ None | string | (mpl\_colors) ] If \textit{None}, the colormap specified by \textit{cmap} will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

\textbf{alpha}: float The alpha blending value

\textbf{cmap}: [ None | Colormap ] A \texttt{cm Colormap} instance or \textit{None}. If \textit{cmap} is \textit{None} and \textit{colors} is \textit{None}, a default Colormap is used.

\textbf{norm}: [ None | Normalize ] A \texttt{matplotlib.colors.Normalize} instance for scaling data values to colors. If \textit{norm} is \textit{None} and \textit{colors} is \textit{None}, the default linear scaling is used.

\textbf{vmin}, \textbf{vmax}: [ None | scalar ] If not \textit{None}, either or both of these values will be supplied to the \texttt{matplotlib.colors.Normalize} instance, overriding the default color scaling based on \textit{levels}.

\textbf{levels}: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass \texttt{levels=[0]}

\textbf{origin}: [ None | ‘upper’ | ‘lower’ | ‘image’ ] If \textit{None}, the first value of \( Z \) will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for \texttt{image.origin} will be used.

This keyword is not active if \( X \) and \( Y \) are specified in the call to contour.

\textbf{extent}: [ None | (x0,x1,y0,y1) ]

If \textit{origin} is not \textit{None}, then \textit{extent} is interpreted as in \texttt{matplotlib.pyplot.imshow()}: it gives the outer pixel boundaries. In this case, the position of \( Z[0,0] \) is the center of the pixel, not a corner. If \textit{origin} is \textit{None}, then \( (x0, y0) \) is the position of \( Z[0,0] \), and \( (x1, y1) \) is the position of \( Z[-1,-1] \).

This keyword is not active if \( X \) and \( Y \) are specified in the call to contour.
locator: [None | ticker.Locator subclass] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the V argument.

extend: [‘neither’ | ‘both’ | ‘min’ | ‘max’] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

xunits, yunits: [None | registered units] Override axis units by specifying an instance of a matplotlib.units.ConversionInterface.

antialiased: [True | False] enable antialiasing, overriding the defaults. For filled contours, the default is True. For line contours, it is taken from rc-Params[‘lines.antialiased’].

contour-only keyword arguments:

linestyles: [None | number | tuple of numbers] If linestyles is None, the default width in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

linestyles: [None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’] If linestyles is None, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the matplotlibrc contour.negative_linestyle setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

nchunk: [0 | integer] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly nchunk by nchunk points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless antialiased is False.

hatches: A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: contourf fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:

z1 < z <= z2

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.

Examples:
Simplest default with labels

```
import matplotlib.pyplot as plt
plt.show()
```
labels at selected locations
Lines with colorbar

-2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0
-3 -2 -1 0 1 2 3

-1.2 -0.8 -0.4 0.0 0.4 0.8 1.2 1.6
Nonsense (3 masked regions)
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotly.pyplot.cool()
```
set the default colormap to cool and apply to current image if any. See help(colormaps) for more information

```
matplotly.pyplot.copper()
```
set the default colormap to copper and apply to current image if any. See help(colormaps) for more information

```
matplotly.pyplot.csd(x, y, NFFT=None, Fs=None, Fc=None, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, return_line=None, **kwargs)
```
Plot the cross-spectral density.

Call signature:
```
 csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, 
 window=mlab.window_hanning, noverlap=0, pad_to=None, 
 sides='default', scale_by_freq=False, return_line=None, **kwargs)
```

The cross spectral density $P_{xy}$ by Welch’s average periodogram method. The vectors $x$ and $y$ are divided into $NFFT$ length segments. Each segment is detrended by function $detrend$ and windowed by function $window$. $noverlap$ gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.
If \(\text{len}(x) < NFFT\) or \(\text{len}(y) < NFFT\), they will be zero padded to \(NFFT\).

\(x, y\): 1-D arrays or sequences Arrays or sequences containing the data

Keyword arguments:

- **Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, \(\text{freqs}\), in cycles per time unit. The default value is 2.

- **window**: callable or ndarray A function or a vector of length \(NFFT\). To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer The number of points to which the data segment is padded when performing the FFT. This can be different from \(NFFT\), which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \(n\) parameter in the call to \(\text{fft()}\). The default is None, which sets pad_to equal to \(NFFT\).

- **NFFT**: integer The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

- **detrend**: [‘default’ | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’] or callable
  The function applied to each segment before \(\text{fft()}\)-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

- **scale_by_freq**: boolean
  Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

- **noverlap**: integer The number of points of overlap between segments. The default value is 0 (no overlap).

- **Fc**: integer The center frequency of \(x\) (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and down-sampled to baseband.

- **return_line**: bool Whether to include the line object plotted in the returned values. Default is False.

If return_line is False, returns the tuple \((Pxy, freqs)\). If return_line is True, returns the tuple \((Pxy, freqs, line)\):

- **Pxy**: 1-D array The values for the cross spectrum \(P_{xy}\) before scaling (complex-valued)
**freqs**: 1-D array  The frequencies corresponding to the elements in $P_{xy}$

**line**: a Line2D instance  The line created by this function. Only returned if `return_line` is True.

For plotting, the power is plotted as $10 \log_{10}(P_{xy})$ for decibels, though $P_{xy}$ itself is returned.


*kwargs* control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-‘</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.13 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

![Graph showing time and frequency data with CSD (db) and psd() function examples.]

See also:

- `psd()` is the equivalent to setting y=x.
- Additional kwargs: hold = [True,False] overrides default hold state

```
import matplotlib.pyplot as plt

# Example usage:
```

```
```
cid = canvas.mpl_connect('button_press_event', on_press)
#...later
canvas.mpl_disconnect(cid)

matplotlib.pyplot.draw()
Redraw the current figure.

This is used in interactive mode to update a figure that has been altered using one or more plot object
method calls; it is not needed if figure modification is done entirely with pyplot functions, if a sequence
of modifications ends with a pyplot function, or if matplotlib is in non-interactive mode and the
sequence of modifications ends with show() or savefig().

A more object-oriented alternative, given any Figure instance, fig, that was created using a pyplot
function, is:

    fig.canvas.draw()

matplotlib.pyplot.errorbar(x, y, yerr=None, xerr=None, fmt='u', ecolor=None, elinewidth=None, capsize=3, barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, hold=None, **kwargs)
Plot an errorbar graph.

Call signature:

    errorbar(x, y, yerr=None, xerr=None, fmt='u', ecolor=None, elinewidth=None, capsize=3, barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, hold=None, **kwargs)

Plot x versus y with error deltas in yerr and xerr. Vertical errorbars are plotted if yerr is not None. Horizontal errorbars are plotted if xerr is not None.

x, y, xerr, and yerr can all be scalars, which plots a single error bar at x, y.

Optional keyword arguments:

    xerr/yerr: [ scalar | N, Nx1, or 2xN array-like ] If a scalar number, len(N) array-like
object, or an Nx1 array-like object, errorbars are drawn at +/-value relative to the
data.

If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the
data.

    fmt: [ ‘’ | ‘none’ | plot format string ] The plot format symbol. If fmt is ‘none’ (case-
insensitive), only the errorbars are plotted. This is used for adding errorbars to a
bar plot, for example. Default is ‘’, an empty plot format string; properties are then
identical to the defaults for plot().

    ecolor: [ None | mpl color ] A matplotlib color arg which gives the color the errorbar
lines; if None, use the color of the line connecting the markers.

    elinewidth: scalar The linewidth of the errorbar lines. If None, use the linewidth.

    capsize: scalar The length of the error bar caps in points

63.1. matplotib.pyplot

1451
capthick: scalar An alias kwarg to markeredgewidth (a.k.a. - mew). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if mew or markeredgewidth are given, then they will over-ride capthick. This may change in future releases.

barsabove: [ True | False ] if True, will plot the errorbars above the plot symbols. Default is below.

lolims / uplims / xlolims / xuplims: [ False | True ] These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. lims-arguments may be of the same type as xerr and yerr. To use limits with inverted axes, set_xlim() or set_ylim() must be called before errorbar().

eroorevery: positive integer subsamples the errorbars. e.g., if everyerror=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

All other keyword arguments are passed on to the plot command for the markers. For example, this code makes big red squares with thick green edges:

```python
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s',
        mfc='red', mec='green', ms=20, mew=4)
```

where mfc, mec, ms and mew are aliases for the longer property names, markerfacecolor, markeredgecolor, markersize and markeredgewidth.

valid kwargs for the marker properties are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestlye or ls</td>
<td>['-'.</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 63.14 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Returns *(plotline, caplines, barlinecols):*

- **plotline**: *Line2D* instance *x, y* plot markers and/or line
- **caplines**: list of error bar cap *Line2D* instances
- **barlinecols**: list of *LineCollection* instances for the horizontal and vertical error ranges.

**Example:**
Additional kwargs: hold = [True]False] overrides default hold state

```
matplotlib.pyplot.eventplot(positions, orientation=u'horizontal', lineoffsets=1, linelengths=1, linewidths=None, colors=None, linestyles=u'solid', hold=None, **kwargs)
```

Plot identical parallel lines at specific positions.

Call signature:

```
eventplot(positions, orientation='horizontal', lineoffsets=0, linelengths=1, linewidths=None, colors=None, linestyles='solid')
```

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

- **orientation** [[‘horizontal’ | ‘vertical’ ]] ‘horizontal’ : the lines will be vertical and arranged in rows
  “vertical” : lines will be horizontal and arranged in columns

- **lineoffsets** : A float or array-like containing floats.
- **linelengths** : A float or array-like containing floats.
**linewidths**: A float or array-like containing floats.

**colors** must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed) or a list of such sequences

**linestyles**: [ 'solid' | 'dashed' | 'dashdot' | 'dotted' ] or an array of these values

For linelengths, linewidths, colors, and linestyles, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as positions, and each value will be applied to the corresponding row or column in positions.

Returns a list of `matplotlib.collections.EventCollection` objects that were added.

**kwargs** are `LineCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiased</td>
<td>Boolean or sequence of boolens</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ '/ '</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with <code>%s</code> conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>segments</td>
<td>unknown</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.15 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>verts</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

```
Additional kwargs: hold = [True]False overrides default hold state

matplotlib.pyplot.figimage(*args, **kwargs)

Adds a non-resampled image to the figure.

call signatures:

figimage(X, **kwargs)

adds a non-resampled array X to the figure.

figimage(X, xo, yo)

with pixel offsets xo, yo,
```
X must be a float array:

- If X is MxN, assume luminance (grayscale)
- If X is MxNx3, assume RGB
- If X is MxNx4, assume RGBA

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xo or yo</td>
<td>An integer, the x and y image offset in pixels</td>
</tr>
<tr>
<td>cmap</td>
<td>a <code>matplotlib.colors.Colormap</code> instance, e.g., <code>cm.jet</code>. If <code>None</code>, default to the <code>rc image cmap</code> value</td>
</tr>
<tr>
<td>norm</td>
<td>a <code>matplotlib.colors.Normalize</code> instance. The default is <code>normalization()</code>. This scales luminance -&gt; 0-1</td>
</tr>
<tr>
<td>vmin</td>
<td><code>vmax</code> used to scale a luminance image to 0-1. If either is <code>None</code>, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for <code>vmin</code> and <code>vmax</code> will be ignored.</td>
</tr>
<tr>
<td>alpha</td>
<td>the alpha blending value, default is <code>None</code></td>
</tr>
<tr>
<td>origin</td>
<td>[ ‘upper’</td>
</tr>
</tbody>
</table>

figimage complements the axes image (imshow()) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an `Axes` with size [0,1,0,1].

An `matplotlib.image.FigureImage` instance is returned.
Additional kwargs are Artist kwargs passed on to FigureImage. Addition kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.figlegend(handles, labels, loc, **kwargs)
Place a legend in the figure.
labels a sequence of strings
handles a sequence of Line2D or Patch instances
loc can be a string or an integer specifying the legend location
A matplotlib.legend.Legend instance is returned.

Example:

figlegend( (line1, line2, line3),
 ('label1', 'label2', 'label3'),
 'upper right' )

See also:

legend()

matplotlib.pyplot.figtext(*args, **kwargs)
Add text to figure.
Call signature:
text(x, y, s, fontdict=None, **kwargs)

Add text to figure at location x, y (relative 0-1 coords). See `text()` for the meaning of the other arguments.

**kwargs control the Text properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
</tbody>
</table>
matplotlib.pyplot.figure

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```
matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, **kwargs)
```

Creates a new figure.

**Parameters**

- **num**: integer or string, optional, default: none
  If not provided, a new figure will be created, and the figure number will be incremented. The figure objects holds this number in a `num` attribute. If num is provided, and a figure with this id already exists, make it active, and returns a reference to it. If this figure does not exist, create it and returns it. If num is a string, the window title will be set to this figure’s `num`.
- **figsize**: tuple of integers, optional, default: None
  width, height in inches. If not provided, defaults to `rc figure.figsize`.
- **dpi**: integer, optional, default: None
  resolution of the figure. If not provided, defaults to `rc figure.dpi`.
- **facecolor**: the background color. If not provided, defaults to `rc figure.facecolor`
- **edgecolor**: the border color. If not provided, defaults to `rc figure.edgecolor`

**Returns**

- **figure**: Figure
  The Figure instance returned will also be passed to `new_figure_manager` in the backends, which allows to hook custom Figure classes into the `pylab` interface. Additional `kwargs` will be passed to the figure `init` function.

**Notes**

If you are creating many figures, make sure you explicitly call “close” on the figures you are not using, because this will enable `pylab` to properly clean up the memory.

`rcParams` defines the default values, which can be modified in the `matplotlibrc` file

```
matplotlib.pyplot.fill(*args, **kwargs)
```

Plot filled polygons.

Call signature:

```
fill(*args, **kwargs)
```

`args` is a variable length argument, allowing for multiple `x, y` pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at `x, y` in blue:
ax.fill(x, y, 'b')

An arbitrary number of x, y, color groups can be specified:

ax.fill(x1, y1, 'g', x2, y2, 'r')

Return value is a list of Patch instances that were added.

The same color strings that plot() supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and y along x, use fill_between()

The closed kwarg will close the polygon when True (default).

kwargs control the Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Example:

```
import matplotlib.pyplot as plt

x = np.linspace(0, 1, 100)
y1 = np.sin(2 * np.pi * x)
y2 = 0.5 * np.sin(4 * np.pi * x)

plt.fill_between(x, y1, y2, where=None, interpolate=False, hold=None, **kwargs)
```

Additional kwargs: hold = [True|False] overrides default hold state

```
plt.fill_between(x, y1, y2=0, where=None, interpolate=False, hold=None, **kwargs)
```

Make filled polygons between two curves.

Call signature:

```
fill_between(x, y1, y2=0, where=None, **kwargs)
```

Create a PolyCollection filling the regions between y1 and y2 where where==True

- **x**: An N-length array of the x data
- **y1**: An N-length array (or scalar) of the y data
- **y2**: An N-length array (or scalar) of the y data
- **where**: If None, default to fill between everywhere. If not None, it is an N-length numpy boolean array and the fill will only happen over the regions where where==True.
- **interpolate**: If True, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the x array.
- **kwargs**: Keyword args passed on to the PolyCollection.

kwargs control the Polygon properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between \( y_1 \) and 0

between \( y_1 \) and 1

between \( y_1 \) and \( y_2 \)
fill between where

Now regions with $y2>1$ are masked
See also:

fill_betweenx() for filling between two sets of x-values

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.fill_betweenx(y, x1, x2=0, where=None, hold=None, **kwargs)

Make filled polygons between two horizontal curves.

Call signature:

fill_betweenx(y, x1, x2=0, where=None, **kwargs)

Create a PolyCollection filling the regions between x1 and x2 where where==True

y : An N-length array of the y data
x1 : An N-length array (or scalar) of the x data
x2 : An N-length array (or scalar) of the x data
where : If None, default to fill between everywhere. If not None, it is a N length numpy
        boolean array and the fill will only happen over the regions where where==True
kwargs : keyword args passed on to the PolyCollection

kwargs control the Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.19 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgcolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>[True</td>
</tr>
<tr>
<td>lod</td>
<td>unknown</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between $y_1$ and 0

between $y_1$ and 1

between $y_1$ and $y_2$
Now regions with y2 > 1 are masked

See also:

- `fill_between()` for filling between two sets of y-values

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.findobj(o=None, match=None, include_self=True)`

Find artist objects.

Recursively find all `Artist` instances contained in self.

`match` can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., `Line2D`. Only return artists of class type.

If `include_self` is True (default), include self in the list to be checked for a match.

`matplotlib.pyplot.flag()`

set the default colormap to flag and apply to current image if any. See `help(colormaps)` for more information

`matplotlib.pyplot.gca(**kwargs)`

Get the current `Axes` instance on the current figure matching the given keyword args, or create one.

See also:

- `matplotlib.figure.Figure.gca` The figure’s gca method.
Examples

To get the current polar axes on the current figure:

```python
plt.gca(projection='polar')
```

If the current axes doesn't exist, or isn't a polar one, the appropriate axes will be created and then returned.

```python
matplotlib.pyplot.gcf()
```

Get a reference to the current figure.

```python
matplotlib.pyplot.gci()
```

Get the current colorable artist. Specifically, returns the current `ScalarMappable` instance (image or patch collection), or `None` if no images or patch collections have been defined. The commands `imshow()` and `figimage()` create `Image` instances, and the commands `pcolor()` and `scatter()` create `Collection` instances. The current image is an attribute of the current axes, or the nearest earlier axes in the current figure that contains an image.

```python
matplotlib.pyplot.get_current_fig_manager()
```

```python
matplotlib.pyplot.get_figlabels()
```

Return a list of existing figure labels.

```python
matplotlib.pyplot.get_fignums()
```

Return a list of existing figure numbers.

```python
matplotlib.pyplot.get_plot_commands()
```

Get a sorted list of all of the plotting commands.

```python
matplotlib.pyplot.ginput(*args, **kwargs)
```

Call signature:

```python
ginput(self, n=1, timeout=30, show_clicks=True,
       mouse_add=1, mouse_pop=3, mouse_stop=2)
```

Blocking call to interact with the figure.

This will wait for \( n \) clicks from the user and return a list of the coordinates of each click.

If `timeout` is zero or negative, does not timeout.

If \( n \) is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons at once) terminates the input.

Right clicking cancels last input.

The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overridden via the arguments `mouse_add`, `mouse_pop` and `mouse_stop`, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.
The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

```
matplotlib.pyplot.gray()
```

set the default colormap to gray and apply to current image if any. See help(colormaps) for more information

```
matplotlib.pyplot.grid(b=None, which=u'major', axis=u'both', **kwargs)
```

Turn the axes grids on or off.

Call signature:

```
grid(self, b=None, which='major', axis='both', **kwargs)
```

Set the axes grids on or off; `b` is a boolean. (For MATLAB compatibility, `b` may also be a string, ‘on’ or ‘off’.)

If `b` is `None` and `len(kwargs)==0`, toggle the grid state. If `kwargs` are supplied, it is assumed that you want a grid and `b` is thus set to `True`.

`which` can be ‘major’ (default), ‘minor’, or ‘both’ to control whether major tick grids, minor tick grids, or both are affected.

`axis` can be ‘both’ (default), ‘x’, or ‘y’ to control which set of gridlines are drawn.

`kwargs` are used to set the grid line properties, e.g.,

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid `Line2D` `kwargs` are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
</tbody>
</table>
Table 63.20 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle or ls</td>
<td>['-', '--', '-.', ':', 'None', '</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True, False]</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None, int, length-2 tuple of int, slice, list/array of int, float, length-2 tuple of float]</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```
import matplotlib.pyplot

matplotlib.pyplot.hexbin(x, y, C=None, gridsize=100, bins=None, xscale='linear', yscale='linear', extent=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors='none', reduce_C_function=<function mean at 0x7f8bfbc90230>, mincnt=None, marginals=False, hold=None, **kwargs)
```

Make a hexagonal binning plot.

Call signature:

```
hexbin(x, y, C=None, gridsize=100, bins=None, xscale='linear', yscale='linear',
 colormap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors='none',
 reduce_C_function=np.mean, mincnt=None, marginals=True, hold=None,
 **kwargs)
```

Make a hexagonal binning plot of x versus y, where x, y are 1-D sequences of the same length, N. If C is None (the default), this is a histogram of the number of occurrences of the observations at (x[i],y[i]).
If $C$ is specified, it specifies values at the coordinate $(x[i],y[i])$. These values are accumulated for each hexagonal bin and then reduced according to $\text{reduce}_C$ function, which defaults to numpy’s mean function ($\text{np.mean}$). (If $C$ is specified, it must also be a 1-D sequence of the same length as $x$ and $y$.)

$x$, $y$ and/or $C$ may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

- **gridsize**: [100 | integer]  
  The number of hexagons in the $x$-direction, default is 100. The corresponding number of hexagons in the $y$-direction is chosen such that the hexagons are approximately regular. Alternatively, gridsize can be a tuple with two elements specifying the number of hexagons in the $x$-direction and the $y$-direction.

- **bins**: [None | ‘log’ | integer | sequence]  
  If None, no binning is applied; the color of each hexagon directly corresponds to its count value.

  If ‘log’, use a logarithmic scale for the color map. Internally, $\log_{10}(i + 1)$ is used to determine the hexagon color.

  If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.

  If a sequence of values, the values of the lower bound of the bins to be used.

- **xscale**: [‘linear’ | ‘log’]  
  Use a linear or log10 scale on the horizontal axis.

- **yscale**: [‘linear’ | ‘log’]  
  Use a linear or log10 scale on the vertical axis.

- **mincnt**: [None | a positive integer]  
  If not None, only display cells with more than mincnt number of points in the cell.

- **marginals**: [True | False]  
  If marginals is True, plot the marginal density as colormapped rectangles along the bottom of the $x$-axis and left of the $y$-axis.

- **extent**: [None | scalars (left, right, bottom, top)]  
  The limits of the bins. The default assigns the limits based on gridsize, $x$, $y$, xscale and yscale.

Other keyword arguments controlling color mapping and normalization arguments:

- **cmap**: [None | Colormap] a matplotlib.colors.Colormap instance. If None, defaults to rc image.cmap.

- **norm**: [None | Normalize] matplotlib.colors.Normalize instance is used to scale luminance data to 0,1.

- **vmin** / **vmax**: scalar  
  vmin and vmax are used in conjunction with norm to normalize luminance data.

  If either are None, the min and max of the color array $C$ is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

- **alpha**: scalar between 0 and 1, or None  
  the alpha value for the patches

- **linestyles**: [None | scalar]  
  If None, defaults to rc lines.linewidth. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by RegularPolyCollection.

Other keyword arguments controlling the Collection properties:

- **edgecolors**: [None | ‘none’ | mpl color | color sequence]  
  If ‘none’, draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.

  If None, draws the outlines in the default color.

  If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the Collection kwargs:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ '/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

The return value is a PolyCollection instance; use get_array() on this PolyCollection to get the counts in each hexagon. If marginals is True, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes hbar and vbar.

Example:
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.hist(x, bins=10, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, hold=None, **kwargs)
```

Plot a histogram.

Compute and draw the histogram of `x`. The return value is a tuple (`n`, `bins`, `patches`) or `([n0, n1, ...], bins, [patches0, patches1,...])` if the input contains multiple data.

Multiple data can be provided via `x` as a list of datasets of potentially different length (`[x0, x1, ...]`), or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form.

Masked arrays are not supported at present.

**Parameters**

- `x` : (n,) array or sequence of (n,) arrays
  - Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length
- `bins` : integer or array_like, optional
  - If an integer is given, `bins + 1` bin edges are returned, consistently with `numpy.histogram()` for numpy version >= 1.3.
  - Unequally spaced bins are supported if `bins` is a sequence.
  - default is 10
range : tuple or None, optional
The lower and upper range of the bins. Lower and upper outliers are
ignored. If not provided, range is (x.min(), x.max()). Range has no
effect if bins is a sequence.

If bins is a sequence or range is specified, autoscaling is based on
the specified bin range instead of the range of x.

Default is None

normed : boolean, optional
If True, the first element of the return tuple will be the counts normal-
ized to form a probability density, i.e., \( \frac{n}{\text{len}(x) \cdot \text{dbin}} \), i.e., the
integral of the histogram will sum to 1. If stacked is also True, the
sum of the histograms is normalized to 1.

Default is False

weights : (n, ) array_like or None, optional
An array of weights, of the same shape as x. Each value in x only
contributes its associated weight towards the bin count (instead of 1).
If normed is True, the weights are normalized, so that the integral of
the density over the range remains 1.

Default is None

cumulative : boolean, optional
If True, then a histogram is computed where each bin gives the counts
in that bin plus all bins for smaller values. The last bin gives the total
number of datapoints. If normed is also True then the histogram is
normalized such that the last bin equals 1. If cumulative evaluates
to less than 0 (e.g., -1), the direction of accumulation is reversed. In
this case, if normed is also True, then the histogram is normalized
such that the first bin equals 1.

Default is False

bottom : array_like, scalar, or None
Location of the bottom baseline of each bin. If a scalar, the base line
for each bin is shifted by the same amount. If an array, each bin is
shifted independently and the length of bottom must match the number
of bins. If None, defaults to 0.

Default is None

histtype : {'bar', 'barstacked', 'step', 'stepfilled'}, optional
The type of histogram to draw.
- 'bar' is a traditional bar-type histogram. If multiple data are
given the bars are aranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are
stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

Default is 'bar'

align : {'left', 'mid', 'right'}, optional
Controls how the histogram is plotted.
• 'left': bars are centered on the left bin edges.
• 'mid': bars are centered between the bin edges.
• 'right': bars are centered on the right bin edges.

Default is 'mid'

orientation : {'horizontal', 'vertical'}, optional
If 'horizontal', barh will be used for bar-type histograms and the bottom kwarg will be the left edges.

rwidth : scalar or None, optional
The relative width of the bars as a fraction of the bin width. If None, automatically compute the width.

Ignored if histtype is 'step' or 'stepfilled'.

Default is None

log : boolean, optional
If True, the histogram axis will be set to a log scale. If log is True and x is a 1D array, empty bins will be filtered out and only the non-empty (n, bins, patches) will be returned.

Default is False

color : color or array_like of colors or None, optional
Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

Default is None

label : string or None, optional
String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

default is None

stacked : boolean, optional
If True, multiple data are stacked on top of each other If False multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

Default is False

Returns n : array or list of arrays
The values of the histogram bins. See normed and weights for a description of the possible semantics. If input x is an array, then this is an array of length nbins. If input is a sequence arrays [data1, data2,...], then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

bins : array
The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches : list or list of lists
Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

Other Parameters kwargs : Patch properties
See also:

`hist2d` 2D histograms

Notes

Until numpy release 1.5, the underlying numpy histogram function was incorrect with `normed='True` if bin sizes were unequal. MPL inherited that error. It is now corrected within MPL when using earlier numpy versions.

Examples

```
40  60  80 100 120 140 160
50  80 110 140 170 200 230

Histogram of IQ: µ = 100, σ = 15
```

Additional kwargs: hold = [True|False] overrides default hold state

```
import matplotlib.pyplot as plt
plt.hist2d(x, y, bins=10, range=None, normed=False, weights=None, cmin=None, cmax=None, hold=None, **kwargs)
```

Make a 2D histogram plot.

Parameters `x, y`: array_like, shape (n,)

Input values

- `bins`: [None | int | [int, int] | array_like | [array, array]]

The bin specification:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If \([\text{int, int}],\) the number of bins in each dimension \((nx, ny = \text{bins})\).
- If \(\text{array_like},\) the bin edges for the two dimensions \((x\_edges=y\_edges=\text{bins})\).
- If \([\text{array, array}],\) the bin edges in each dimension \((x\_edges, y\_edges = \text{bins})\).

The default value is 10.

**range** : array_like shape(2, 2), optional, default: None
The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): \([[\text{xmin, xmax}], [\text{ymin, ymax}]]\). All values outside of this range will be considered outliers and not tallied in the histogram.

**normed** : boolean, optional, default: False
Normalize histogram.

**weights** : array_like, shape (n, ), optional, default: None
An array of values \(w\_i\) weighing each sample \((x\_i, y\_i)\).

**cmin** : scalar, optional, default: None
All bins that have count less than \(\text{cmin}\) will not be displayed and these count values in the return value count histogram will also be set to \(\text{nan}\) upon return.

**cmax** : scalar, optional, default: None
All bins that have count more than \(\text{cmax}\) will not be displayed (set to \(\text{none}\) before passing to imshow) and these count values in the return value count histogram will also be set to \(\text{nan}\) upon return.

**Returns**
The return value is \((\text{counts, xedges, yedges, Image})\).

**Other Parameters**
**kwargs** : pcolorfast() properties.

**See also:**

**hist** 1D histogram

**Notes**

Rendering the histogram with a logarithmic color scale is accomplished by passing a `colors.LogNorm` instance to the `norm` keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with `colors.PowerNorm`. 
Examples

```
matplotlib.pyplot.hlines(y, xmin, xmax, colors='k', linestyles='solid', label=''
hold=None, **kwargs)
```

Plot horizontal lines at each y from xmin to xmax.

**Parameters**
- `y`: scalar or sequence of scalar y-indexes where to plot the lines.
- `xmin, xmax`: scalar or 1D array_like
  - Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- `colors`: array_like of colors, optional, default: ‘k’
- `linestyles`: [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’], optional
- `label`: string, optional, default: ‘’

**Returns**
- `lines`: LineCollection

**Other Parameters**
- `kwargs`: LineCollection properties.

See also:
- `vlines` vertical lines
Examples

Additional kwargs: hold = [True|False] overrides default hold state

**matplotlib.pyplot.hold**(b=``None``)

Set the hold state. If `b` is None (default), toggle the hold state, else set the hold state to boolean value `b`:

```
hold()        # toggle hold
hold(True)   # hold is on
hold(False)  # hold is off
```

When `hold` is True, subsequent plot commands will be added to the current axes. When `hold` is False, the current axes and figure will be cleared on the next plot command.

**matplotlib.pyplot.hot()**

set the default colormap to hot and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.hsv()**

set the default colormap to hsv and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.imread(**args**, **kwargs**)**

Read an image from a file into an array.

`fname` may be a string path or a Python file-like object. If using a file object, it must be opened in binary mode.

If `format` is provided, will try to read file of that type, otherwise the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Return value is a numpy.array. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.
matplotlib can only read PNGs natively, but if PIL is installed, it will use it to load the image and return an array (if possible) which can be used with `imshow()`.

```python
matplotlib.pyplot.imsave(*args, **kwargs)
```

Save an array as in image file.

The output formats available depend on the backend being used.

**Arguments:**

- `fname`: A string containing a path to a filename, or a Python file-like object. If `format` is `None` and `fname` is a string, the output format is deduced from the extension of the filename.
- `arr`: An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

**Keyword arguments:**

- `vmin/vmax`: A [None | scalar] that set the color scaling for the image by fixing the values that map to the colormap color limits. If either `vmin` or `vmax` is `None`, that limit is determined from the `arr` min/max value.
- `cmap`: cmap is a colors.Colormap instance, e.g., cm.jet. If None, default to the rc image.cmap value.
- `format`: One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.
- `origin`: [‘upper’ | ‘lower’] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value.
- `dpi`: The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

```python
matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, hold=None, **kwargs)
```

Display an image on the axes.
If `interpolation` is `None`, default to `rc` `image.interpolation`. See also the `filternorm` and `filterrad` parameters. If `interpolation` is `'none'`, then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to `'nearest'`.

**norm**: `Normalize`, optional, default: `None`

A `Normalize` instance is used to scale luminance data to 0, 1. If `None`, use the default function: `normalize`. `norm` is only used if `X` is an array of floats.

**vmin, vmax**: scalar, optional, default: `None`

`vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

**alpha**: scalar, optional, default: `None`

The alpha blending value, between 0 (transparent) and 1 (opaque)

**origin**: `[upper | lower]`, optional, default: `None`

Place the [0,0] index of the array in the upper left or lower left corner of the axes. If `None`, default to `rc` `image.origin`.

**extent**: scalars (left, right, bottom, top), optional, default: `None`

The location, in data-coordinates, of the lower-left and upper-right corners. If `None`, the image is positioned such that the pixel centers fall on zero-based (row, column) indices.

**shape**: scalars (columns, rows), optional, default: `None`

For raw buffer images

**filternorm**: scalar, optional, default: 1

A parameter for the antigrain image resize filter. From the antigrain documentation, if `filternorm` = 1, the filter normalizes integer values and corrects the rounding errors. It doesn’t do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

**filterrad**: scalar, optional, default: 4.0

The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'

**Returns image**: `AxesImage`

**Other Parameters** `kwargs`: `Artist` properties.

**See also**:

`matshow` Plot a matrix or an array as an image.
matplotlib.pyplot.ion()  
Turn interactive mode on.

matplotlib.pyplot.ishold()  
Return the hold status of the current axes.

matplotlib.pyplot.isinteractive()  
Return status of interactive mode.

matplotlib.pyplot.jet()  
set the default colormap to jet and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.legend(*args, **kwargs)  
Places a legend on the axes.

To make a legend for lines which already exist on the axes (via plot for instance), simply call this function with an iterable of strings, one for each legend item. For example:
ax.plot([1, 2, 3])
ax.legend(['A simple line'])

However, in order to keep the “label” and the legend element instance together, it is preferable to specify the label either at artist creation, or by calling the `set_label()` method on the artist:

```python
line, = ax.plot([1, 2, 3], label='Inline label')
# Overwrite the label by calling the method.
line.set_label('Label via method')
ax.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling `legend()` without any arguments and without setting the labels manually will result in no legend being drawn.

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```python
legend((line1, line2, line3), ('label1', 'label2', 'label3'))
```

**Parameters**

- `loc`: int or string or pair of floats, default: 0
  The location of the legend. Possible codes are:

<table>
<thead>
<tr>
<th>Location String</th>
<th>Location Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘best’</td>
<td>0</td>
</tr>
<tr>
<td>‘upper right’</td>
<td>1</td>
</tr>
<tr>
<td>‘upper left’</td>
<td>2</td>
</tr>
<tr>
<td>‘lower left’</td>
<td>3</td>
</tr>
<tr>
<td>‘lower right’</td>
<td>4</td>
</tr>
<tr>
<td>‘right’</td>
<td>5</td>
</tr>
<tr>
<td>‘center left’</td>
<td>6</td>
</tr>
<tr>
<td>‘center right’</td>
<td>7</td>
</tr>
<tr>
<td>‘lower center’</td>
<td>8</td>
</tr>
<tr>
<td>‘upper center’</td>
<td>9</td>
</tr>
<tr>
<td>‘center’</td>
<td>10</td>
</tr>
</tbody>
</table>

  Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates (in which case `bbox_to_anchor` will be ignored).

- `bbox_to_anchor`: matplotlib.transforms.BboxBase instance or tuple of floats
  Specify any arbitrary location for the legend in bbox_transform coordinates (default Axes coordinates).

  For example, to put the legend’s upper right hand corner in the center of the axes the following keywords can be used:

  ```python
  loc='upper right', bbox_to_anchor=(0.5, 0.5)
  ```

- `ncol`: integer
  The number of columns that the legend has. Default is 1.
prop : None or matplotlib.font_manager.FontProperties or dict
The font properties of the legend. If None (default), the current matplotlib.rcParams will be used.

fontsize : int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}
Controls the font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if prop is not specified.

numpoints : None or int
The number of marker points in the legend when creating a legend entry for a line/matplotlib.lines.Line2D. Default is None which will take the value from the legend.numpoints rcParam.

scatterpoints : None or int
The number of marker points in the legend when creating a legend entry for a scatter plot/ matplotlib.collections.PathCollection. Default is None which will take the value from the legend.scatterpoints rcParam.

scatteryoffsets : iterable of floats
The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5]. Default [0.375, 0.5, 0.3125].

markerscale : None or int or float
The relative size of legend markers compared with the originally drawn ones. Default is None which will take the value from the legend.markerscale rcParam.

frameon : None or bool
Control whether a frame should be drawn around the legend. Default is None which will take the value from the legend.frameon rcParam.

fancybox : None or bool
Control whether round edges should be enabled around the FancyBboxPatch which makes up the legend’s background. Default is None which will take the value from the legend.fancybox rcParam.

shadow : None or bool
Control whether to draw a shadow behind the legend. Default is None which will take the value from the legend.shadow rcParam.

framealpha : None or float
Control the alpha transparency of the legend’s frame. Default is None which will take the value from the legend.framealpha rcParam.

mode : {"expand", None}
If mode is set to "expand" the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor if defines the legend’s size).

bbox_transform : None or matplotlib.transforms.Transform
The transform for the bounding box (bbox_to_anchor). For a value of None (default) the Axes’ transAxes transform will be used.

title : str or None
The legend’s title. Default is no title (None).

**borderpad**: float or None
The fractional whitespace inside the legend border. Measured in font-size units. Default is None which will take the value from the legend.borderpad rcParam.

**labelspacing**: float or None
The vertical space between the legend entries. Measured in font-size units. Default is None which will take the value from the legend.labelspacing rcParam.

**handlelength**: float or None
The length of the legend handles. Measured in font-size units. Default is None which will take the value from the legend.handlelength rcParam.

**handletextpad**: float or None
The pad between the legend handle and text. Measured in font-size units. Default is None which will take the value from the legend.handletextpad rcParam.

**borderaxespad**: float or None
The pad between the axes and legend border. Measured in font-size units. Default is None which will take the value from the legend.borderaxespad rcParam.

**columnspacing**: float or None
The spacing between columns. Measured in font-size units. Default is None which will take the value from the legend.columnspacing rcParam.

**handler_map**: dict or None
The custom dictionary mapping instances or types to a legend handler. This handler_map updates the default handler map found at matplotlib.legend.Legend.get_legend_handler_map().

**Notes**

Not all kinds of artist are supported by the legend command. See Legend guide for details.
Examples

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot([0, 0.5, 1, 1.5, 2, 2.5, 3], [0, 5, 10, 15, 20, 25, 0])
ax.set_xlabel('Model length')
ax.set_ylabel('Data length')
ax.set_title('Total message length')

ax.plot([0, 0.5, 1, 1.5, 2, 2.5, 3], [0, 5, 10, 15, 20, 25, 0], '--', label='Model length')
ax.plot([0, 0.5, 1, 1.5, 2, 2.5, 3], [0, 5, 10, 15, 20, 25, 0], ':', label='Data length')
ax.plot([0, 0.5, 1, 1.5, 2, 2.5, 3], [0, 5, 10, 15, 20, 25, 0], '-.', label='Total message length')

ax.legend()
```

```
matplotlib.pyplot.locator_params(axis='both', tight=None, **kwargs)

Control behavior of tick locators.

Keyword arguments:

axis ['x' | 'y' | 'both'] Axis on which to operate; default is 'both'.

tight [True | False | None] Parameter passed to autoscale_view(). Default is None, for no change.

Remaining keyword arguments are passed to directly to the set_params() method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:

```
ax.locator_params(tight=True, nbins=4)
```

Because the locator is involved in autoscaling, autoscale_view() is called automatically after the parameters are changed.

This presently works only for the MaxNLocator used by default on linear axes, but it may be generalized.

```
matplotlib.pyplot.loglog(*args, **kwargs)

Make a plot with log scaling on both the x and y axis.

Call signature:
```
loglog(*args, **kwargs)

loglog() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

- **basex/basey:** scalar > 1 Base of the x/y logarithm
- **subsx/subsy:** [None | sequence] The location of the minor x/y ticks; None defaults to autosubs, which depend on the number of decades in the plot; see matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale() for details
- **nonposx/nonposy:** ['mask' | 'clip'] Non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.22 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

![Graphs showing various plots with labels](image)

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.magnitude_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, scale=None, hold=None, **kwargs)
```

Plot the magnitude spectrum.
Call signature:

```
magnitude_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
    pad_to=None, sides='default', **kwargs)
```

Compute the magnitude spectrum of `x`. Data is padded to a length of `pad_to` and the windowing function `window` is applied to the signal.

`x`: 1-D array or sequence  
Array or sequence containing the data

Keyword arguments:

`Fs`: scalar  
The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

`window`: callable or ndarray  
A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides`: [‘default’ | ‘onesided’ | ‘twosided’]  
Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

`pad_to`: integer  
The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `n` parameter in the call to `fft()`. The default is None, which sets `pad_to` equal to the length of the input signal (i.e. no padding).

`scale`: [‘default’ | ‘linear’ | ‘dB’]  
The scaling of the values in the `spec`. ‘linear’ is no scaling. ‘dB’ returns the values in dB scale. When `mode` is ‘density’, this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). ‘default’ is ‘linear’.

`Fc`: integer  
The center frequency of `x` (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and down-sampled to baseband.

Returns the tuple `(spectrum, freqs, line)`:

`spectrum`: 1-D array  
The values for the magnitude spectrum before scaling (real valued)

`freqs`: 1-D array  
The frequencies corresponding to the elements in `spectrum`

`line`: a Line2D instance  
The line created by this function

kwarg control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.23 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
See also:

- `psd()` plots the power spectral density.
- `angle_spectrum()` plots the angles of the corresponding frequencies.
- `phase_spectrum()` plots the phase (unwrapped angle) of the corresponding frequencies.
- `specgram()` can plot the magnitude spectrum of segments within the signal in a colormap.

Additional kwargs: `hold = [True|False]` overrides default hold state

```python
matplotlib.pyplot.margins(*args, **kw)
```
Set or retrieve autoscaling margins.

signatures:

- `margins()`
  returns `xmargin`, `ymargin`

- `margins(margin)`

- `margins(x=xmargin, y=ymargin)`

- `margins(x=xmargin, y=ymargin)`

63.1. matplotlib.pyplot
margins(..., tight=False)

All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. The tight parameter is passed to autoscale_view(), which is executed after a margin is changed; the default here is True, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting tight to None will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if xmargin is not None, then xmargin times the X data interval will be added to each end of that interval before it is used in autoscaling.

matplotlib.pyplot.matshow(A, fignum=None, **kw)

Display an array as a matrix in a new figure window.

The origin is set at the upper left hand corner and rows (first dimension of the array) are displayed horizontally. The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

With the exception of fignum, keyword arguments are passed to imshow(). You may set the origin kwarg to “lower” if you want the first row in the array to be at the bottom instead of the top.

fignum: [ None | integer | False ] By default, matshow() creates a new figure window with automatic numbering. If fignum is given as an integer, the created figure will use this figure number. Because of how matshow() tries to set the figure aspect ratio to be the one of the array, if you provide the number of an already existing figure, strange things may happen.

If fignum is False or 0, a new figure window will NOT be created.

matplotlib.pyplot.minorticks_off()

Remove minor ticks from the current plot.

matplotlib.pyplot.minorticks_on()

Display minor ticks on the current plot.

Displaying minor ticks reduces performance; turn them off using minorticks_off() if drawing speed is a problem.

matplotlib.pyplot.over(func, *args, **kwargs)

Call a function with hold(True).

Calls:

func(*args, **kwargs)

with hold(True) and then restores the hold state.

matplotlib.pyplot.pause(interval)

Pause for interval seconds.

If there is an active figure it will be updated and displayed, and the GUI event loop will run during the pause.

If there is no active figure, or if a non-interactive backend is in use, this executes time.sleep(interval).
This can be used for crude animation. For more complex animation, see `matplotlib.animation`.

This function is experimental; its behavior may be changed or extended in a future release.

```python
matplotlib.pyplot.pcolor(*args, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

**Note:** `pcolor` can be very slow for large arrays; consider using the similar but much faster `pcolormesh()` instead.

Call signatures:

```python
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

*C* is the array of color values.

*X* and *Y*, if given, specify the \((x, y)\) coordinates of the colored quadrilaterals; the quadrilateral for *C*[i,j] has corners at:

\[
(X[i, j], Y[i, j]),
(X[i, j+1], Y[i, j+1]),
(X[i+1, j], Y[i+1, j]),
(X[i+1, j+1], Y[i+1, j+1]).
\]

Ideally the dimensions of *X* and *Y* should be one greater than those of *C*; if the dimensions are the same, then the last row and column of *C* will be ignored.

Note that the the column index corresponds to the \(x\)-coordinate, and the row index corresponds to \(y\); for details, see the \textit{Grid Orientation} section below.

If either or both of *X* and *Y* are 1-D arrays or column vectors, they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

*X*, *Y* and *C* may be masked arrays. If either *C*[i, j], or one of the vertices surrounding *C*[i,j] (*X* or *Y* at [i, j], [i+1, j], [i, j+1],[i+1, j+1]) is masked, nothing is plotted.

Keyword arguments:

\begin{itemize}
\item \texttt{cmap} [ 
\texttt{None} | \texttt{Colormap} ] A `matplotlib.colors.Colormap` instance. If \texttt{None}, use rc settings.
\item \texttt{norm} [ 
\texttt{None} | \texttt{Normalize} ] An `matplotlib.colors.Normalize` instance is used to scale luminance data to 0.1. If \texttt{None}, defaults to `normalize()`.
\item \texttt{vmin/vmax} [ 
\texttt{None} | \texttt{scalar} ] `vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either is \texttt{None}, it is autoscaled to the respective min or max of the color array *C*. If not \texttt{None}, `vmin` or `vmax` passed in here override any pre-existing values supplied in the \texttt{norm} instance.
\item \texttt{shading} [ \texttt{‘flat’} | \texttt{‘faceted’} ] If ‘faceted’, a black grid is drawn around each rectangle; if ‘flat’, edges are not drawn. Default is ‘flat’, contrary to MATLAB. \textbf{This kwarg is deprecated; please use ‘edgecolors’ instead:}
\begin{itemize}
\item shading=’flat’ – edgecolors=’none’
\item shading=’faceted’ – edgecolors=’k’
\end{itemize}
\end{itemize}
**edgecolors:** [ None | 'none' | color | color sequence] If None, the rc setting is used by default.

If 'none', edges will not be visible.

An mpl color or sequence of colors will set the edge color

**alpha:** $0 \leq \text{scalar} \leq 1$ or None the alpha blending value

**snap:** bool Whether to snap the mesh to pixel boundaries.

Return value is a matplotlib.collections.Collection instance. The grid orientation follows the MATLAB convention: an array $C$ with shape $(\text{nrows}, \text{ncolumns})$ is plotted with the column number as $X$ and the row number as $Y$, increasing up; hence it is plotted the way the array would be printed, except that the $Y$ axis is reversed. That is, $C$ is taken as $C^*(*y, x)$.

Similarly for meshgrid():

```python
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```

is equivalent to:

```python
X = array(
[[0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4]])
Y = array(
[[0, 0, 0, 0, 0],
 [1, 1, 1, 1, 1],
 [2, 2, 2, 2, 2]])
```

so if you have:

```python
C = rand(len(x), len(y))
```

then you need to transpose C:

```python
pcolor(X, Y, C.T)
```

or:

```python
pcolor(C.T)
```

MATLAB `pcolor()` always discards the last row and column of $C$, but matplotlib displays the last row and column if $X$ and $Y$ are not specified, or if $X$ and $Y$ have one more row and column than $C$.

**kwargs can be used to control the PolyCollection properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.24 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>offsets</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Note: The default antialiaseds is False if the default edgecolors*="none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors is not “none”, then the default antialiaseds is taken from rcParams[‘patch.antialiased’], which defaults to True. Stroking the edges may be preferred if alpha is 1, but will cause artifacts otherwise.

See also:

pcolormesh() For an explanation of the differences between pcolor and pcolormesh.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.pcolormesh(*args, **kwargs)
Plot a quadrilateral mesh.

Call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

pcolormesh is similar to `pcolor()`, but uses a different mechanism and returns a different object; `pcolor` returns a `PolyCollection` but `pcolormesh` returns a `QuadMesh`. It is much faster, so it is almost always preferred for large arrays.

`C` may be a masked array, but `X` and `Y` may not. Masked array support is implemented via `cmap` and `norm`; in contrast, `pcolor()` simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

- **`cmap`**: `[None | Colormap]` A `matplotlib.colors.Colormap` instance. If `None`, use `rc` settings.
- **`norm`**: `[None | Normalize]` A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If `None`, defaults to `normalize()`.
- **`vmin/vmax`**: `[None | scalar]` `vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either is `None`, it is autoscaled to the respective min or max of the color array `C`. If not `None`, `vmin` or `vmax` passed in here override any pre-existing values supplied in the `norm` instance.
- **`shading`**: `[‘flat’ | ‘gouraud’]` ‘flat’ indicates a solid color for each quad. When ‘gouraud’, each quad will be Gouraud shaded. When gouraud shading, edgecolors is ignored.
- **`edgecolors`**:
  - `[None | ‘None’ | ‘face’ | color | color sequence]`
  - If `None`, the `rc` setting is used by default.
  - If ‘None’, edges will not be visible.
  - If ‘face’, edges will have the same color as the faces.

An mpl color or sequence of colors will set the edge color

- **`alpha`**: `0 <= scalar <= 1 or None` the alpha blending value

Return value is a `matplotlib.collections.QuadMesh` object.

`kwargs` can be used to control the `matplotlib.collections.QuadMesh` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or <code>None</code></td>
</tr>
<tr>
<td><code>animated</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>antialiased or antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.25 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>[True</td>
</tr>
<tr>
<td>lod</td>
<td>unknown</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

`pcolor()` For an explanation of the grid orientation and the expansion of 1-D X and/or Y to 2-D arrays.

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.phase_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning, pad_to=None, sides='default', hold=None, **kwargs)
```

Plot the phase spectrum.

Call signature:

```python
phase_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning, pad_to=None, sides='default', **kwargs)
```

Compute the phase spectrum (unwrapped angle spectrum) of x. Data is padded to a length of pad_to and the windowing function window is applied to the signal.
**x: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

- **Fs:** scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window:** callable or ndarray A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides:** [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to:** integer The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

- **Fc:** integer The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and down-sampled to baseband.

Returns the tuple (spectrum, freqs, line):

- **spectrum:** 1-D array The values for the phase spectrum in radians (real valued)
- **freqs:** 1-D array The frequencies corresponding to the elements in spectrum
- **line:** a Line2D instance The line created by this function

**kwargs control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>((Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 63.26 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[’-’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[’butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[’miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Example:**
See also:

- `magnitude_spectrum()` plots the magnitudes of the corresponding frequencies.
- `angle_spectrum()` plots the wrapped version of this function.
- `specgram()` can plot the phase spectrum of segments within the signal in a colormap.

Additional kwargs:

- `hold` = [True|False] overrides default hold state

Matplotlib.pyplot.pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1, startangle=None, radius=None, counterclock=True, wedgeprops=None, textprops=None)

Plot a pie chart.

Call signature:

```
pie(x, explode=None, labels=None, colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'), autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1, startangle=None, radius=None, counterclock=True, wedgeprops=None, textprops=None)
```

Make a pie chart of array x. The fractional area of each wedge is given by x/sum(x). If sum(x) <= 1, then the values of x give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the x-axis.
Keyword arguments:

- **explode**: [None | len(x) sequence] If not None, is a len(x) array which specifies the fraction of the radius with which to offset each wedge.

- **colors**: [None | color sequence] A sequence of matplotlib color args through which the pie chart will cycle.

- **labels**: [None | len(x) sequence of strings] A sequence of strings providing the labels for each wedge

- **autopct**: [None | format string | format function] If not None, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be fmt%pct. If it is a function, it will be called.

- **pctdistance**: scalar The ratio between the center of each pie slice and the start of the text generated by autopct. Ignored if autopct is None; default is 0.6.

- **labelldistance**: scalar The radial distance at which the pie labels are drawn

- **shadow**: [False | True] Draw a shadow beneath the pie.

- **startangle**: [None | Offset angle] If not None, rotates the start of the pie chart by angle degrees counterclockwise from the x-axis.

- **radius**: [None | scalar] The radius of the pie, if radius is None it will be set to 1.

- **counterclock**: [False | True] Specify fractions direction, clockwise or counterclockwise.

- **wedgeprops**: [None | dict of key value pairs] Dict of arguments passed to the wedge objects making the pie. For example, you can pass in wedgeprops = {‘linewidth’ : 3} to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default clip_on=False.

- **textprops**: [None | dict of key value pairs] Dict of arguments to pass to the text objects.

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. e.g.:

```python
figure(figsize=(8,8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

or:

```python
axes(aspect=1)
```

**Return value:** If autopct is None, return the tuple (patches, texts):

- patches is a sequence of matplotlib.patches.Wedge instances
- texts is a list of the label matplotlib.text.Text instances.

If autopct is not None, return the tuple (patches, texts, autotexts), where patches and texts are as above, and autotexts is a list of Text instances for the numeric labels.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.pink()

set the default colormap to pink and apply to current image if any. See help(colormaps) for more information
matplotlib.pyplot.plot(*args, **kwargs)

Plot lines and/or markers to the Axes. args is a variable length argument, allowing for multiple x, y pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)         # plot x and y using default line style and color
plot(x, y, 'bo')   # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')      # ditto, but with red plus
```

If x and/or y is 2-dimensional, then the corresponding columns will be plotted.

An arbitrary number of x, y, fmt groups can be specified, as in:

```
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different color specified by a ‘color cycle’. To change this behavior, you can edit the axes.color_cycle rcParam.

The following format string characters are accepted to control the line style or marker:

<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>solid line style</td>
</tr>
<tr>
<td>'--'</td>
<td>dashed line style</td>
</tr>
<tr>
<td>'-.'</td>
<td>dash-dot line style</td>
</tr>
<tr>
<td>':'</td>
<td>dotted line style</td>
</tr>
<tr>
<td>','</td>
<td>point marker</td>
</tr>
<tr>
<td>'o'</td>
<td>circle marker</td>
</tr>
<tr>
<td>'v'</td>
<td>triangle-down marker</td>
</tr>
<tr>
<td>'^'</td>
<td>triangle-up marker</td>
</tr>
<tr>
<td>'&lt;'</td>
<td>triangle-left marker</td>
</tr>
<tr>
<td>'&gt;'</td>
<td>triangle-right marker</td>
</tr>
<tr>
<td>'1'</td>
<td>tri_down marker</td>
</tr>
<tr>
<td>'2'</td>
<td>tri_up marker</td>
</tr>
<tr>
<td>'3'</td>
<td>tri_left marker</td>
</tr>
<tr>
<td>'4'</td>
<td>tri_right marker</td>
</tr>
<tr>
<td>'s'</td>
<td>square marker</td>
</tr>
<tr>
<td>'p'</td>
<td>pentagon marker</td>
</tr>
<tr>
<td>'*'</td>
<td>star marker</td>
</tr>
<tr>
<td>'h'</td>
<td>hexagon1 marker</td>
</tr>
<tr>
<td>'H'</td>
<td>hexagon2 marker</td>
</tr>
<tr>
<td>'+'</td>
<td>plus marker</td>
</tr>
<tr>
<td>'x'</td>
<td>x marker</td>
</tr>
<tr>
<td>'D'</td>
<td>diamond marker</td>
</tr>
<tr>
<td>'d'</td>
<td>thin_diamond marker</td>
</tr>
<tr>
<td>'1'</td>
<td>vline marker</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
</tbody>
</table>

The following color abbreviations are supported:
In addition, you can specify colors in many weird and wonderful ways, including full names (‘green’), hex strings (‘#008000’), RGB or RGBA tuples ((0,1,0,1)) or grayscale intensities as a string (‘0.8’). Of these, the string specifications can be used in place of a fmt group, but the tuple forms can be used only as kwargs.

Line styles and colors are combined in a single format string, as in ’bo’ for blue circles.

The kwargs can be used to set line properties (any property that has a set_* method). You can use this to set a line label (for auto legends), linewidth, antialiased, marker face color, etc. Here is an example:

```python
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs', label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```python
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and markerfacecolor with:

```python
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See Line2D for details.

The kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased  or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63.27 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_path</td>
<td>([Path, Transform]</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

*kwargs scalex and scaley, if defined, are passed on to autoscale_view() to determine whether the x and y axes are autoscaled; the default is True.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.plot_date(x, y, fmt='u’o’, tz=None, xdate=True, ydate=False, hold=None, **kwargs)

Plot with data with dates.

Call signature:
plot_date(x, y, fmt='bo', tz=None, xdate=True,
ydate=False, **kwargs)

Similar to the plot() command, except the x or y (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or y can be a sequence of dates represented as float days since 0001-01-01 UTC.

Keyword arguments:

  **fmt:** string  The plot format string.
  **tz:** [ None | timezone string | tzinfo instance] The time zone to use in labeling dates.
    If None, defaults to rc value.
  **xdate:** [ True | False ] If True, the x-axis will be labeled with dates.
  **ydate:** [ False | True ] If True, the y-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set
the formatters/locators after the call to plot_date() since plot_date() will set the
default tick locator to matplotlib.dates.AutoDateLocator (if the tick locator is not al-
ready set to a matplotlib.dates.DateLocator instance) and the default tick formatter
to matplotlib.dates.AutoDateFormatter (if the tick formatter is not already set to a
matplotlib.dates.DateFormatter instance).

Valid kwarg are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgedwidth or mew</td>
<td>float value in points</td>
</tr>
</tbody>
</table>
See also:

dates for helper functions
date2num(), num2date() and drange() for help on creating the required floating point dates.

Additional kwargs: hold = [True]False] overrides default hold state

matplotlib.pyplot.plotfile(fname, cols=(0,), plotfuncs=None, comments=u'\#', skiprows=0,
checkrows=5, delimiter=u',', names=None, subplots=True, new-fig=True, **kwargs)

Plot the data in in a file.

cols is a sequence of column identifiers to plot. An identifier is either an int or a string. If it is an int, it indicates the column number. If it is a string, it indicates the column header. matplotlib will make column headers lower case, replace spaces with underscores, and remove all illegal characters; so 'Adj Close*' will have name 'adj_close'.

• If len(cols) == 1, only that column will be plotted on the y axis.
• If len(cols) > 1, the first element will be an identifier for data for the x axis and the remaining elements will be the column indexes for multiple subplots if subplots is True (the default), or for lines in a single subplot if subplots is False.

plotfuncs, if not None, is a dictionary mapping identifier to an Axes plotting function as a string. Default is 'plot', other choices are 'semilogy', 'fill', 'bar', etc. You must use the same type of identifier in the cols vector as you use in the plotfuncs dictionary, e.g., integer column numbers in both or column names in both. If subplots is False, then including any function such as 'semilogy' that changes the axis scaling will set the scaling for all columns.

comments, skiprows, checkrows, delimiter, and names are all passed on to
matplotlib.pylab.csv2rec() to load the data into a record array.

If `newfig` is `True`, the plot always will be made in a new figure; if `False`, it will be made in the current figure if one exists, else in a new figure.

`kwargs` are passed on to plotting functions.

Example usage:

```python
# plot the 2nd and 4th column against the 1st in two subplots
plotfile(fname, (0,1,3))

# plot using column names; specify an alternate plot type for volume
plotfile(fname, ('date', 'volume', 'adj_close'),
         plotfuncs={'volume': 'semilogy'})
```

Note: `plotfile` is intended as a convenience for quickly plotting data from flat files; it is not intended as an alternative interface to general plotting with `pyplot` or `matplotlib`.

```python
matplotlib.pyplot.polar(*args, **kwargs)
```

Make a polar plot.

call signature:

```python
polar(theta, r, **kwargs)
```

Multiple `theta`, `r` arguments are supported, with format strings, as in `plot()`.

```python
matplotlib.pyplot.prism()
```

set the default colormap to prism and apply to current image if any. See help(colormaps) for more information

```python
matplotlib.pyplot.psd(x, NFFT=None, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, return_line=None, **kwargs)
```

Plot the power spectral density.

Call signature:

```python
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, return_line=None, **kwargs)
```

The power spectral density \( P_{xx} \) by Welch’s average periodogram method. The vector `x` is divided into `NFFT` length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The \( |\text{fft}(i)|^2 \) of each segment \( i \) are averaged to compute \( P_{xx} \), with a scaling to correct for power loss due to windowing.

If `len(x) < NFFT`, it will be zero padded to `NFFT`.

- `x`: 1-D array or sequence  Array or sequence containing the data

Keyword arguments:

- `Fs`: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit. The default value is 2.
window: callable or ndarray A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides: ['default' | 'onesided' | 'twosided'] Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

pad_to: integer The number of points to which the data segment is padded when performing the FFT. This can be different from NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to NFFT

NFFT: integer The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

detrend: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or callable
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

scale_by_freq: boolean
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap: integer The number of points of overlap between segments. The default value is 0 (no overlap).

Fc: integer The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and down-sampled to baseband.

return_line: bool Whether to include the line object plotted in the returned values. Default is False.

If return_line is False, returns the tuple (Pxx, freqs). If return_line is True, returns the tuple (Pxx, freqs, line):

Pxx: 1-D array The values for the power spectrum P_{xx} before scaling (real valued)
freqs: 1-D array The frequencies corresponding to the elements in Pxx
line: a Line2D instance The line created by this function. Only return if return_line is True.

For plotting, the power is plotted as 10 \log_{10}(P_{xx}) for decibels, though P_{xx} itself is returned.

kwargs control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>ID array</td>
</tr>
<tr>
<td>ydata</td>
<td>ID array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
See also:

`specgram()` differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

`magnitude_spectrum()` plots the magnitude spectrum.

`csd()` plots the spectral density between two signals.

Additional kwargs: `hold = [True|False]` overrides default hold state

```python
matplotlib.pyplot.quiver(*args, **kw)
```

Plot a 2-D field of arrows.

call signatures:

```python
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

Arguments:

- `X, Y`: The x and y coordinates of the arrow locations (default is tail of arrow; see `pivot` kwarg)
- `U, V`: Give the x and y components of the arrow vectors
- `C`: An optional array used to map colors to the arrows
All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if len(X) and len(Y) match the column and row dimensions of U, then X and Y will be expanded with numpy.meshgrid().

U, V, C may be masked arrays, but masked X, Y are not supported at present.

Keyword arguments:

- **units**: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy'] Arrow units; the arrow dimensions except for length are in multiples of this unit.
  - 'width' or 'height': the width or height of the axes
  - 'dots' or 'inches': pixels or inches, based on the figure dpi
  - 'x', 'y', or 'xy': X, Y, or sqrt(X^2+Y^2) data units

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width or 'height', the arrow size increases with the width and height of the axes, respectively, when the the window is resized; for ‘dots’ or ‘inches’, resizing does not change the arrows.

- **angles**: ['uv' | 'xy' | array] With the default ‘uv’, the arrow axis aspect ratio is 1, so that if U==V the orientation of the arrow on the plot is 45 degrees CCW from the horizontal axis (positive to the right). With ‘xy’, the arrow points from (x,y) to (x+u, y+v). Use this for plotting a gradient field, for example. Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the horizontal axis. Note: inverting a data axis will correspondingly invert the arrows only with angles='xy'.

- **scale**: [None | float] Data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. If None, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the scale_units parameter.

- **scale_units**: None, or any of the units options. For example, if scale_units is ‘inches’, scale is 2.0, and (u,v) = (1,0), then the vector will be 0.5 inches long. If scale_units is ‘width’, then the vector will be half the width of the axes.

If scale_units is ‘x’ then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use “angles='xy', scale_units='xy', scale=1’”.

- **width**: Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

- **headwidth**: scalar Head width as multiple of shaft width, default is 3

- **headlength**: scalar Head length as multiple of shaft width, default is 5

- **headaxislength**: scalar Head length at shaft intersection, default is 4.5

- **minshaft**: scalar Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

- **minlength**: scalar Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

- **pivot**: ['tail' | 'middle' | 'tip'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot.

- **color**: [ color | color sequence ] This is a synonym for the PolyCollection facecolor kwarg. If C has been set, color has no effect.

The defaults give a slightly swept-back arrow; to make the head a triangle, make headaxislength the...
same as `headlength`. To make the arrow more pointed, reduce `headwidth` or increase `headlength` and `headaxislength`. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave `minshaft` alone.

Linewidths and edgecolors can be used to customize the arrow outlines. Additional `PolyCollection` keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[ '/'</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>linestyles</code> or <code>dashes</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code> or <code>linewidths</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

Additional kwargs: `hold = [True|False]` overrides default hold state

```
import matplotlib.pyplot as plt

plt.quiverkey(*args, **kw)
```
Add a key to a quiver plot.

Call signature:

```python
quiverkey(Q, X, Y, U, label, **kw)
```

Arguments:

- `Q`: The Quiver instance returned by a call to `quiver`.
- `X, Y`: The location of the key; additional explanation follows.
- `U`: The length of the key
- `label`: A string with the length and units of the key

Keyword arguments:

- `coordinates` = [‘axes’ | ‘figure’ | ‘data’ | ‘inches’] Coordinate system and units for `X`, `Y`: ‘axes’ and ‘figure’ are normalized coordinate systems with 0,0 in the lower left and 1,1 in the upper right; ‘data’ are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); ‘inches’ is position in the figure in inches, with 0,0 at the lower left corner.
- `color`: overrides face and edge colors from `Q`.
- `labelpos` = [‘N’ | ‘S’ | ‘E’ | ‘W’] Position the label above, below, to the right, to the left of the arrow, respectively.
- `labelsep`: Distance in inches between the arrow and the label. Default is 0.1
- `labelcolor`: defaults to default `Text` color.
- `fontproperties`: A dictionary with keyword arguments accepted by the `FontProperties` initializer: `family`, `style`, `variant`, `size`, `weight`

Any additional keyword arguments are used to override vector properties taken from `Q`.

The positioning of the key depends on `X, Y, coordinates, and labelpos`. If `labelpos` is ‘N’ or ‘S’, `X, Y` give the position of the middle of the key arrow. If `labelpos` is ‘E’, `X, Y` positions the head, and if `labelpos` is ‘W’, `X, Y` positions the tail; in either of these two cases, `X, Y` is somewhere in the middle of the arrow+label key object.

Additional kwargs: `hold = [True|False]` overrides default hold state

```python
matplotlib.pyplot.rc(*args, **kwargs)
```

Set the current rc params. Group is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., `(xтик, yтик). kwargs is a dictionary attribute name/value pairs, e.g.,:

```python
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```python
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:
Thus you could abbreviate the above rc command as:

```python
rc('lines', lw=2, c='r')
```

Note you can use python’s kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```python
font = {'family': 'monospace',
        'weight': 'bold',
        'size': 'larger'}

rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `rcdefaults()` to restore the default rc params after changes.

```python
matplotlib.pyplot.rc_context(rc=None, fname=None)
```

Return a context manager for managing rc settings.

This allows one to do:

```python
with mpl.rc_context(fname='screen.rc'):
    plt.plot(x, a)

with mpl.rc_context(fname='print.rc'):
    plt.plot(x, b)
    plt.plot(x, c)
```

The ‘a’ vs ‘x’ and ‘c’ vs ‘x’ plots would have settings from ‘screen.rc’, while the ‘b’ vs ‘x’ plot would have settings from ‘print.rc’.

A dictionary can also be passed to the context manager:

```python
with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
    plt.plot(x, a)
```

The ‘rc’ dictionary takes precedence over the settings loaded from ‘fname’. Passing a dictionary only is also valid.

```python
matplotlib.pyplot.rcdefaults()
```

Restore the default rc params. These are not the params loaded by the rc file, but mpl’s internal params. See `rc_file_defaults` for reloading the default params from the rc file.
matplotlib.pyplot.rgrids(*args, **kwargs)
Get or set the radial gridlines on a polar plot.

call signatures:

lines, labels = rgrids()
lines, labels = rgrids(radii, labels=None, angle=22.5, **kwargs)

When called with no arguments, rgrid() simply returns the tuple (lines, labels), where lines is an array of radial gridlines (Line2D instances) and labels is an array of tick labels (Text instances). When called with arguments, the labels will appear at the specified radial distances and angles.

labels, if not None, is a len(radii) list of strings of the labels to use at each angle.

If labels is None, the rformatter will be used

Examples:

# set the locations of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0) )

# set the locations and labels of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0), (‘Tom’, ‘Dick’, ‘Harry’) )

matplotlib.pyplot.savefig(*args, **kwargs)
Save the current figure.

Call signature:

savefig(fname, dpi=None, facecolor=’w’, edgecolor=’w’, 
orientation=’portrait’, papertype=None, format=None, 
transparent=False, bbox_inches=None, pad_inches=0.1, 
frameon=None)

The output formats available depend on the backend being used.

Arguments:

fname: A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as PdfPages.

If format is None and fname is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter savefig.format is used.

If fname is not a string, remember to specify format to ensure that the correct backend is used.

Keyword arguments:

dpi: [ None | scalar > 0 ] The resolution in dots per inch. If None it will default to the value savefig.dpi in the matplotlibrc file.

facecolor, edgecolor: the colors of the figure rectangle

orientation: [ ‘landscape’ | ‘portrait’ ] not supported on all backends; currently only on postscript output

**format:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

**transparent:** If True, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

**frameon:** If True, the figure patch will be colored, if False, the figure background will be transparent. If not provided, the rcParam ‘savefig.frameon’ will be used.

**bbox_inches:** Bbox in inches. Only the given portion of the figure is saved. If ‘tight’, try to figure out the tight bbox of the figure.

**pad_inches:** Amount of padding around the figure when bbox_inches is ‘tight’.

**bbox_extra_artists:** A list of extra artists that will be considered when the tight bbox is calculated.

```python
matplotlib.pyplot.sca(ax)
```

Set the current Axes instance to `ax`.

The current Figure is updated to the parent of `ax`.

```python
matplotlib.pyplot.scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, hold=None, **kwargs)
```

Make a scatter plot of `x` vs `y`, where `x` and `y` are sequence like objects of the same lengths.

**Parameters**

- **x, y:** array_like, shape (n, )
  - Input data
- **s:** scalar or array_like, shape (n, ), optional, default: 20
  - size in points^2.
- **c:** color or sequence of color, optional, default
  - `c` can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the `cmap` and `norm` specified via kwargs (see below). Note that `c` should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. `c` can be a 2-D array in which the rows are RGB or RGBA, however.
- **marker:** `MarkerStyle`, optional, default: ‘o’
  - See `markers` for more information on the different styles of markers scatter supports.
- **cmap:** `Colormap`, optional, default: None
  - A `Colormap` instance or registered name. `cmap` is only used if `c` is an array of floats. If None, defaults to `rc.image.cmap`.
- **norm:** `Normalize`, optional, default: None
  - A `Normalize` instance is used to scale luminance data to 0, 1. `norm` is only used if `c` is an array of floats. If None, use the default `normalize()`.
- **vmin, vmax:** scalar, optional, default: None
  - `vmin` and `vmax` are used in conjunction with `norm` to normalize lumi-
nance data. If either are None, the min and max of the color array is
used. Note if you pass a norm instance, your settings for vmin and
vmax will be ignored.

alpha : scalar, optional, default: None
The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths : scalar or array_like, optional, default: None
If None, defaults to (lines.linewidth,). Note that this is a tuple, and if
you set the linewidths argument you must set it as a sequence of floats,
as required by RegularPolyCollection.

Returns paths: PathCollection

Other Parameters kwargs: Collection properties

Notes

Any or all of x, y, s, and c may be masked arrays, in which case all masks will be combined and only
unmasked points will be plotted.

Examples

Additional kwargs: hold = [True|False] overrides default hold state
matplotlib.pyplot.sci(im)
Set the current image. This image will be the target of colormap commands like jet(), hot() or clim(). The current image is an attribute of the current axes.

matplotlib.pyplot.semilogx(*args, **kwargs)
Make a plot with log scaling on the x axis.

Call signature:

semilogx(*args, **kwargs)

semilogx() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale().

Notable keyword arguments:

- **basex**: scalar > 1 Base of the x logarithm
- **subsx**: [ None | sequence ] The location of the minor xticks; None defaults to autosubs, which depend on the number of decades in the plot; see set_xscale() for details.
- **nonposx**: [ ‘mask’ | ‘clip’ ] Non-positive values in x can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘-’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>
Table 63.31 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

loglog() For example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.semilogy(*args, **kwargs)

Make a plot with log scaling on the y axis.

call signature:

semilogy(*args, **kwargs)

semilogy() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

basey: scalar > 1 Base of the y logarithm

subsy: [ None | sequence ] The location of the minor yticks; None defaults to autosubs, which depend on the number of decades in the plot; see set_yscale() for details.

nonposy: [ ‘mask’ | ‘clip’ ] Non-positive values in y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 63.32 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

loglog() For example code and figure

Additional kwargs: hold = [True]False overrides default hold state

matplotlib.pyplot.set_cmap(cmap)
Set the default colormap. Applies to the current image if any. See help(colormaps) for more information.

`cmap` must be a Colormap instance, or the name of a registered colormap.

See `matplotlib.cm.register_cmap()` and `matplotlib.cm.get_cmap()`.

```python
Matplotlib.pyplot.setp(*args, **kwargs)
```

Set a property on an artist object.

matplotlib supports the use of `setp()` ("set property") and `getp()` to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```python
>>> line, = plot([1,2,3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```python
>>> setp(line, 'linestyle')
linestyle: ['-' | '--' | '-.' | ':' | 'steps' | 'None']
```

If you want to see all the properties that can be set, and their possible values, you can do:

```python
>>> setp(line)
... long output listing omitted
```

`setp()` operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```python
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

`setp()` works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```python
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')        # python style
```

```python
Matplotlib.pyplot.show(*args, **kw)
```

Display a figure. When running in ipython with its pylab mode, display all figures and return to the ipython prompt.

In non-interactive mode, display all figures and block until the figures have been closed; in interactive mode it has no effect unless figures were created prior to a change from non-interactive to interactive mode (not recommended). In that case it displays the figures but does not block.
A single experimental keyword argument, block, may be set to True or False to override the blocking behavior described above.

```
matplotlib.pyplot.specgram(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, cmap=None, xextent=None, pad_to=None, sides=None, scale_by_freq=None, mode=None, scale=None, vmin=None, vmax=None, hold=None, **kwargs)
```

Plot a spectrogram.

Call signature:

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=128, cmap=None, xextent=None, pad_to=None, sides='default', scale_by_freq=None, mode='default', scale='default', **kwargs)
```

Compute and plot a spectrogram of data in x. Data are split into NFFT length segments and the spectrum of each section is computed. The windowing function window is applied to each segment, and the amount of overlap of each segment is specified with noverlap. The spectrogram is plotted as a colormap (using imshow).

**x:** 1-D array or sequence  Array or sequence containing the data

Keyword arguments:

- **Fs:** scalar  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window:** callable or ndarray  A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides:** [ 'default' | 'onesided' | 'twosided' ]  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

- **pad_to:** integer  The number of points to which the data segment is padded when performing the FFT. This can be different from NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to NFFT.

- **NFFT:** integer  The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

- **detrend:** [ 'default' | 'constant' | 'mean' | 'linear' | 'none' ] or callable  The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string
to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

**scale_by_freq**: boolean
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**mode**: [ ‘default’ | ‘psd’ | ‘magnitude’ | ‘angle’ | ‘phase’ ] What sort of spectrum to use. Default is ‘psd’. which takes the power spectral density. ‘complex’ returns the complex-valued frequency spectrum. ‘magnitude’ returns the magnitude spectrum. ‘angle’ returns the phase spectrum without unwrapping. ‘phase’ returns the phase spectrum with unwrapping.

**noverlap**: integer The number of points of overlap between blocks. The default value is 128.

**scale**: [ ‘default’ | ‘linear’ | ‘dB’ ] The scaling of the values in the spec. ‘linear’ is no scaling. ‘dB’ returns the values in dB scale. When **mode** is ‘psd’, this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). ‘default’ is ‘dB’ if **mode** is ‘psd’ or ‘magnitude’ and ‘linear’ otherwise. This must be ‘linear’ if **mode** is ‘angle’ or ‘phase’.

**Fc**: integer The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**cmap**: A matplotlib.colors.Colormap instance; if None, use default determined by rc

**xextent**: The image extent along the x-axis. xextent = (xmin,xmax) The default is (0,max(bins)), where bins is the return value from specgram()

**kwarg**: Additional kwargs are passed on to imshow which makes the specgram image

---

**Note:** detrend and scale_by_freq only apply when **mode** is set to ‘psd’

Returns the tuple (spectrum, freqs, t, im):

**spectrum**: 2-D array columns are the periodograms of successive segments

**freqs**: 1-D array The frequencies corresponding to the rows in spectrum

**t**: 1-D array The times corresponding to midpoints of segments (i.e the columns in spectrum)

**im**: instance of class AxesImage The image created by imshow containing the spectrogram

**Example:**
See also:

`psd()` differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

`magnitude_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘magnitude’. Plots a line instead of a colormap.

`angle_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘angle’. Plots a line instead of a colormap.

`phase_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘phase’. Plots a line instead of a colormap.

Additional kwargs: `hold` = [True|False] overrides default hold state

```
import matplotlib.pyplot as plt

# Example usage

fig, (ax1, ax2) = plt.subplots(2, 1)

# Compute the spectrum
frequencies, spectrum = plt.psd(your_data)

# Plot the magnitude spectrum
ax1.magnitude_spectrum(spectrum)

# Plot the angle spectrum
ax2.angle_spectrum(spectrum)

plt.show()
```

### Additional Information

- `matplotlib.pyplot.spectral()`
  - set the default colormap to spectral and apply to current image if any. See help(colormaps) for more information

- `matplotlib.pyplot.spring()`
  - set the default colormap to spring and apply to current image if any. See help(colormaps) for more information

- `matplotlib.pyplot.spy(Z, precision=0, marker=None, markersize=None, aspect='equal', hold=None, **kwargs)`
  - Plot the sparsity pattern on a 2-D array.

  `spy(Z)` plots the sparsity pattern of the 2-D array Z.
**Parameters** $Z$ : sparse array (n, m)

The array to be plotted.

`precision` : float, optional, default: 0

If `precision` is 0, any non-zero value will be plotted; else, values of $|Z| > precision$ will be plotted.

For `scipy.sparse.spmatrix` instances, there is a special case: if `precision` is `present`, any value present in the array will be plotted, even if it is identically zero.

`origin` : ['upper', ‘lower’], optional, default: ‘upper’

Place the [0,0] index of the array in the upper left or lower left corner of the axes.

`aspect` : ['auto' | 'equal' | scalar], optional, default: ‘equal’

If ‘equal’, and `extent` is `None`, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

If ‘auto’, changes the image aspect ratio to match that of the axes.

If `None`, default to rc image.aspect value.

**Two plotting styles are available: image or marker. Both are available for full arrays, but only the marker style works for :class:`scipy.sparse.spmatrix` instances.**

If *marker* and *markersize* are *None*, an image will be returned and any remaining kwargs are passed to :func:`matplotlib.pyplot.imshow`; else, a :class:`matplotlib.lines.Line2D` object will be returned with the value of marker determining the marker type, and any remaining kwargs passed to the :meth:`~matplotlib.axes.Axes.plot` method.

If *marker* and *markersize* are *None*, useful kwargs include:

* *cmap*
* *alpha*

See also:

`imshow` for image options.

`plot` for plotting options

Additional

`matplotlib.pyplot.stackplot(x, *args, **kwargs)`

Draws a stacked area plot.

$x$ : 1d array of dimension N
y [2d array of dimension MxN, OR any number 1d arrays each of dimension] 1xN. The
data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y) # where y is MxN
stackplot(x, y1, y2, y3, y4) # where y1, y2, y3, y4, are all 1xNm
```

Keyword arguments:
```
baseline ["zero", "sym", "wiggle", "weighted_wiggle"] Method used to calculate the
baseline. ‘zero’ is just a simple stacked plot. ‘sym’ is symmetric around zero
and is sometimes called ThemeRiver. ‘wiggle’ minimizes the sum of the squared
slopes. ‘weighted_wiggle’ does the same but weights to account for size of
each layer. It is also called Streamgraph-layout. More details can be found at
http://www.leebyron.com/else/streamgraph/.
```
```
colors [A list or tuple of colors. These will be cycled through and] used to colour the
stacked areas. All other keyword arguments are passed to fill_between()
```

Returns r: A list of PolyCollection, one for each element in the stacked area plot.

Note that Legend does not support PolyCollection objects. To create a legend on a
stackplot, use a proxy artist: http://matplotlib.org/users/legend_guide.html#using-proxy-
artist

Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.stem(*args, **kwargs)
```

Create a stem plot.

Call signatures:
```
stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')
stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')
```

A stem plot plots vertical lines (using linefmt) at each x location from the baseline to y, and places a
marker there using markerfmt. A horizontal line at 0 is is plotted using basefmt.

If no x values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (markerline, stemlines, baseline).

See also:

This document for details.

Example:
Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.step(x, y, *args, **kwargs)
```

Make a step plot.

Call signature:

```python
step(x, y, *args, **kwargs)
```

Additional keyword args to `step()` are the same as those for `plot()`.

`x` and `y` must be 1-D sequences, and it is assumed, but not checked, that `x` is uniformly increasing.

Keyword arguments:

- `where` : ['pre' | 'post' | 'mid'] If ‘pre’, the interval from `x[i]` to `x[i+1]` has level `y[i+1]`
- If ‘post’, that interval has level `y[i]`
- If ‘mid’, the jumps in `y` occur half-way between the `x`-values.

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.streamplot(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None, arrowselect='max', arclength=0.2, minlength=0.1, transform=None, zorder=1, hold=None)
```

Draws streamlines of a vector flow.

`x, y` [1d arrays] an *evenly spaced* grid.
$u, v$ [2d arrays] x and y-velocities. Number of rows should match length of y, and the number of columns should match x.

density [float or 2-tuple] Controls the closeness of streamlines. When density = 1, the domain is divided into a 30x30 grid—density linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use [density_x, density_y].

linewidth [numeric or 2d array] vary linewidth when given a 2d array with the same shape as velocities.

color [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, color values are converted to colors using cmap.

cmap [Colormap] Colormap used to plot streamlines and arrows. Only necessary when using an array input for color.

norm [Normalize] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when color is an array.

arrowsize [float] Factor scale arrow size.


zorder [int] any number

Returns:

stream_container [StreamplotSet] Container object with attributes

- lines: matplotlib.collections.LineCollection of streamlines
- arrows: collection of matplotlib.patches.FancyArrowPatch objects representing arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.subplot(*args, **kwargs)

Return a subplot axes positioned by the given grid definition.

Typical call signature:

subplot(nrows, ncols, plot_number)

Where nrows and ncols are used to notionally split the figure into nrows * ncols sub-axes, and plot_number is used to identify the particular subplot that this function is to create within the notional grid. plot_number starts at 1, increments across rows first and has a maximum of nrows * ncols.

In the case when nrows, ncols and plot_number are all less than 10, a convenience exists, such that the a 3 digit number can be given instead, where the hundreds represent nrows, the tens represent ncols and the units represent plot_number. For instance:

subplot(211)

produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 row by 1 column notional grid (no grid actually exists, but conceptually this is how the returned subplot has been positioned).
Note: Creating a new subplot with a position which is entirely inside a pre-existing axes will trigger the larger axes to be deleted:

```python
import matplotlib.pyplot as plt

# plot a line, implicitly creating a subplot(111)
plt.plot([1,2,3])

# now create a subplot which represents the top plot of a grid
# with 2 rows and 1 column. Since this subplot will overlap the
# first, the plot (and its axes) previously created, will be removed
plt.subplot(211)
plt.plot(range(12))
plt.subplot(212, axisbg='y')  # creates 2nd subplot with yellow background
```

If you do not want this behavior, use the `add_subplot()` method or the `axes()` function instead.

Keyword arguments:

- `axisbg`: The background color of the subplot, which can be any valid color specifier. See `matplotlib.colors` for more information.
- `polar`: A boolean flag indicating whether the subplot plot should be a polar projection. Defaults to `False`.
- `projection`: A string giving the name of a custom projection to be used for the subplot. This projection must have been previously registered. See `matplotlib.projections`.

See also:

- `axes()` For additional information on `axes()` and `subplot()` keyword arguments.
- `examples/pie_and_polar_charts/polar_scatter_demo.py` For an example

Example:
mathplotlib.pyplot.subplot2grid(shape, loc, rowspan=1, colspan=1, **kwargs)

Create a subplot in a grid. The grid is specified by shape, at location of loc, spanning rowspan, colspan cells in each direction. The index for loc is 0-based.

subplot2grid(shape, loc, rowspan=1, colspan=1)

is identical to

gs = GridSpec(shape[0], shape[2])
subplotspec = gs.new_subplotspec(loc, rowspan, colspan)
subplot(subplotspec)

mathplotlib.pyplot.subplot_tool(targetfig=None)

Launch a subplot tool window for a figure.

A matplotlib.widgets.SubplotTool instance is returned.

mathplotlib.pyplot.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot_kw=None, gridspec_kw=None, **fig_kw)

Create a figure with a set of subplots already made.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Keyword arguments:

- **nrows** [int] Number of rows of the subplot grid. Defaults to 1.
**ncols** [int] Number of columns of the subplot grid. Defaults to 1.

**sharex** [string or bool] If True, the X axis will be shared amongst all subplots. If True and you have multiple rows, the x tick labels on all but the last row of plots will have visible set to False. If a string must be one of “row”, “col”, “all”, or “none”. “all” has the same effect as True, “none” has the same effect as False. If “row”, each subplot row will share a X axis. If “col”, each subplot column will share a X axis and the x tick labels on all but the last row will have visible set to False.

**sharey** [string or bool] If True, the Y axis will be shared amongst all subplots. If True and you have multiple columns, the y tick labels on all but the first column of plots will have visible set to False. If a string must be one of “row”, “col”, “all”, or “none”. “all” has the same effect as True, “none” has the same effect as False. If “row”, each subplot row will share a Y axis and the y tick labels on all but the first column will have visible set to False. If “col”, each subplot column will share a Y axis.

**squeeze** [bool] If True, extra dimensions are squeezed out from the returned axis object:

- if only one subplot is constructed (nrows=ncols=1), the resulting single Axis object is returned as a scalar.
- for Nx1 or 1xN subplots, the returned object is a 1-d numpy object array of Axis objects are returned as numpy 1-d arrays.
- for NxM subplots with N>1 and M>1 are returned as a 2d array.

If False, no squeezing at all is done: the returned axis object is always a 2-d array containing Axis instances, even if it ends up being 1x1.

**subplot_kw** [dict] Dict with keywords passed to the add_subplot() call used to create each subplots.

**gridspec_kw** [dict] Dict with keywords passed to the GridSpec constructor used to create the grid the subplots are placed on.

**fig_kw** [dict] Dict with keywords passed to the figure() call. Note that all keywords not recognized above will be automatically included here.

Returns:

fig, ax : tuple

- fig is the matplotlib.figure.Figure object
- ax can be either a single axis object or an array of axis objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

Examples:

```python
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Just a figure and one subplot
f, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Two subplots, unpack the output array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)
```
# Four polar axes
plt.subplots(2, 2, subplot_kw=dict(polar=True))

# Share a X axis with each column of subplots
plt.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
plt.subplots(2, 2, sharey='row')

# Share a X and Y axis with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')
# same as
plt.subplots(2, 2, sharex=True, sharey=True)

matplotlib.pyplot.subplots_adjust(*args, **kwargs)
Tune the subplot layout.

call signature:

subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)

The parameter meanings (and suggested defaults) are:

left = 0.125  # the left side of the subplots of the figure
right = 0.9   # the right side of the subplots of the figure
bottom = 0.1  # the bottom of the subplots of the figure
top = 0.9     # the top of the subplots of the figure
wspace = 0.2  # the amount of width reserved for blank space between subplots
hspace = 0.2  # the amount of height reserved for white space between subplots

The actual defaults are controlled by the rc file

matplotlib.pyplot.summer()
set the default colormap to summer and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.suptitle(*args, **kwargs)
Add a centered title to the figure.

kwags are matplotlib.text.Text properties. Using figure coordinates, the defaults are:
x [0.5] The x location of the text in figure coords
y [0.98] The y location of the text in figure coords
horizontalalignment ['center'] The horizontal alignment of the text
verticalalignment ['top'] The vertical alignment of the text

A matplotlib.text.Text instance is returned.

Example:

fig.suptitle('this is the figure title', fontsize=12)
matplotlib.pyplot.switch_backend(newbackend)

Switch the default backend. This feature is experimental, and is only expected to work switching to an image backend. e.g., if you have a bunch of PostScript scripts that you want to run from an interactive ipython session, you may want to switch to the PS backend before running them to avoid having a bunch of GUI windows popup. If you try to interactively switch from one GUI backend to another, you will explode.

Calling this command will close all open windows.

matplotlib.pyplot.table(**kwargs)

Add a table to the current axes.

Call signature:

table(cellText=None, cellColours=None,
   cellLoc='right', colWidths=None,
   rowLabels=None, rowColours=None, rowLoc='left',
   colLabels=None, colColours=None, colLoc='center',
   loc='bottom', bbox=None):

Returns a matplotlib.table.Table instance. For finer grained control over tables, use the Table class and add it to the axes with add_table().

Thanks to John Gill for providing the class and table.

kwargs control the Table properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontsize</td>
<td>a float in points</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Add text to the axes.

Add text in string `s` to axis at location `x, y`, data coordinates.

**Parameters**

- `x, y`: scalars
data coordinates
- `s`: string
text
- `fontdict`: dictionary, optional, default: None
A dictionary to override the default text properties. If `fontdict` is None, the defaults are determined by your rc parameters.
- `withdash`: boolean, optional, default: False
Creates a `TextWithDash` instance instead of a `Text` instance.

**Other Parameters**

- `kwargs`: `Text` properties.
Other miscellaneous text parameters.

**Examples**

Individual keyword arguments can be used to override any given parameter:

```python
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```python
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...       verticalalignment='center',
...       transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `bbox`. `bbox` is a dictionary of `Rectangle` properties. For example:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

Get or set the theta locations of the gridlines in a polar plot.

If no arguments are passed, return a tuple `(lines, labels)` where `lines` is an array of radial gridlines (`Line2D` instances) and `labels` is an array of tick labels (`Text` instances):

```python
lines, labels = thetagrids()
```

Otherwise the syntax is:

```python
lines, labels = thetagrids(angles, labels=None, fmt='%d', frac = 1.1)
```
set the angles at which to place the theta grids (these gridlines are equal along the theta dimension). 

`angles` is in degrees.

`labels`, if not `None`, is a len(angles) list of strings of the labels to use at each angle.

If `labels` is `None`, the labels will be `fmt%angle`.

`frac` is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples `(lines, labels):`
- `lines` are `Line2D` instances
- `labels` are `Text` instances.

Note that on input, the `labels` argument is a list of strings, and on output it is a list of `Text` instances.

Examples:

```python
# set the locations of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90) )
```

```python
# set the locations and labels of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90), ('NE', 'NW', 'SW', 'SE') )
```

```python
ax.tick_params(axis='both', **kwargs)
```

Change the appearance of ticks and tick labels.

Keyword arguments:
- `axis` `[['x' | 'y' | 'both']]` Axis on which to operate; default is ‘both’.
- `reset` `[[True | False]]` If True, set all parameters to defaults before processing other keyword arguments.  
  Default is `False`.
- `which` `[['major' | 'minor' | 'both']]` Default is ‘major’; apply arguments to which ticks.
- `direction` `[['in' | 'out' | 'inout']]` Puts ticks inside the axes, outside the axes, or both.
- `length` Tick length in points.
- `width` Tick width in points.
- `color` Tick color; accepts any mpl color spec.
- `pad` Distance in points between tick and label.
- `labelsize` Tick label font size in points or as a string (e.g., ‘large’).
- `labelcolor` Tick label color; mpl color spec.
- `colors` Changes the tick color and the label color to the same value: mpl color spec.
- `zorder` Tick and label zorder.
- `bottom`, `top`, `left`, `right` `[[bool | 'on' | 'off']` controls whether to draw the respective ticks.
- `labelbottom`, `labeltop`, `labelleft`, `labelright` Boolean or `['on' | 'off']`, controls whether to draw the respective tick labels.

Example:

```python
ax.tick_params(direction='out', length=6, width=2, colors='r')
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red.
matplotlib.pyplot.ticklabel_format(**kwargs)
Change the ScalarFormatter used by default for linear axes.

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>style</strong></td>
<td>[‘sci’ (or ‘scientific’)]</td>
</tr>
<tr>
<td><strong>scilimits</strong></td>
<td>(m, n), pair of integers; if <em>style</em> is ‘sci’, scientific notation will be used for numbers outside the range 10^{-m} \cdot \text{sup}: to 10^{-n} \cdot \text{sup}.. Use (0,0) to include all numbers.</td>
</tr>
<tr>
<td><strong>useOffset</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>axis</strong></td>
<td>[‘x’</td>
</tr>
<tr>
<td><strong>useLocale</strong></td>
<td>If True, format the number according to the current locale. This affects things such as the character used for the decimal separator. If False, use C-style (English) formatting. The default setting is controlled by the axes.formatter.useLocale rcparam.</td>
</tr>
</tbody>
</table>

Only the major ticks are affected. If the method is called when the ScalarFormatter is not the Formatter being used, an AttributeError will be raised.

matplotlib.pyplot.tight_layout(pad=1.08, h_pad=None, w_pad=None, rect=None)
Automatically adjust subplot parameters to give specified padding.

Parameters:
- **pad** [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.
- **h_pad**, **w_pad** [float] padding (height/width) between edges of adjacent subplots. Defaults to pad_inches.
- **rect** [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

matplotlib.pyplot.title(s, *args, **kwargs)
Set a title of the current axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

See also:
See text() for adding text to the current axes

Parameters:
- **label** :str
  Text to use for the title
- **fontdict** :dict
  A dictionary controlling the appearance of the title text, the default fontdict is
  ```
  {'fontsize': rcParams[\'axes.titlesize\'], ‘fontweight’ : rcParams[\'axes.titleweight\'], ‘verticalalignment’: ‘baseline’, ‘horizontalalignment’: loc}
  ```
- **loc** :[‘center’, ‘left’, ‘right’], str, optional
  Which title to set, defaults to ‘center’

Returns:
- **text** :Text
The matplotlib text instance representing the title

**Other Parameters** \texttt{kwargs} : text properties

Other keyword arguments are text properties, see \texttt{Text} for a list of valid text properties.

\texttt{matplotlib.pyplot.tricontour(*args, **kwargs)}

Draw contours on an unstructured triangular grid. \texttt{tricontour()} and \texttt{tricontourf()} draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

\texttt{tricontour(triangulation, ...)}

where triangulation is a \texttt{matplotlib.tri.Triangulation} object, or

\texttt{tricontour(x, y, ...)}
\texttt{tricontour(x, y, triangles, ...)}
\texttt{tricontour(x, y, triangles=triangles, ...)}
\texttt{tricontour(x, y, mask=mask, ...)}
\texttt{tricontour(x, y, triangles, mask=mask, ...)}

in which case a Triangulation object will be created. See \texttt{Triangulation} for a explanation of these possibilities.

The remaining arguments may be:

\texttt{tricontour(..., Z)}

where \(Z\) is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

\texttt{tricontour(..., Z, N)}

count \(N\) automatically-chosen levels.

\texttt{tricontour(..., Z, V)}

draw contour lines at the values specified in sequence \(V\)

\texttt{tricontourf(..., Z, V)}

fill the \((\text{len}(V)-1)\) regions between the values in \(V\)

\texttt{tricontour(Z, **kwargs)}

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

\(C = \text{tricontour}(...\)) returns a \texttt{TriContourSet} object.

Optional keyword arguments:
colors: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

alpha: float The alpha blending value

cmap: [ None | Colormap ] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

norm: [ None | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

levels [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass levels=[0]

origin: [ None | ‘upper’ | ‘lower’ | ‘image’ ] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if X and Y are specified in the call to contour.

extent: [ None | (x0,x1,y0,y1) ]

If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If origin is None, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1].

This keyword is not active if X and Y are specified in the call to contour.

locator: [ None | ticker.Locator subclass ] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the V argument.

extend: [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

xunits, yunits: [ None | registered units ] Override axis units by specifying an instance of a matplotlib.units.ConversionInterface.

tricontour-only keyword arguments:

lineweight: [ None | number | tuple of numbers ] If lineweight is None, the default width in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this lineweight.

If a tuple, different levels will be plotted with different lineweights in the order specified

linestyles: [ None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If linestyles is None, the ‘solid’ is used.
*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in *contour.negative_linestyle* in *matplotlibrc* will be used.

**tricontourf-only keyword arguments:**

- **antialiased**: [True | False] enable antialiasing
- **nchunk**: [0 | integer] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*. 

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

\[ z_1 < z \leq z_2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

**Examples:**
Contour plot of user-specified triangulation

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.tricontourf(*args, **kwargs)

Draw contours on an unstructured triangular grid. tricontour() and tricontourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

tricontour(triangulation, ...)

where triangulation is a matplotlib.tri.Triangulation object, or

tricontour(x, y, ...)  
tricontour(x, y, triangles, ...)  
tricontour(x, y, triangles=triangles, ...)  
tricontour(x, y, mask=mask, ...)  
tricontour(x, y, triangles, mask=mask, ...)

in which case a Triangulation object will be created. See Triangulation for a explanation of these possibilities.

The remaining arguments may be:
tricontour(..., Z)

where \( Z \) is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

tricontour(..., Z, N)

countour \( N \) automatically-chosen levels.

tricontour(..., Z, V)

draw contour lines at the values specified in sequence \( V \)

tricontourf(..., Z, V)

fill the (len(\( V \))-1) regions between the values in \( V \)

tricontour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

C = tricontour(...) returns a TriContourSet object.

Optional keyword arguments:

- **colors**: [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.
  
  If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

  If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- **alpha**: float The alpha blending value

- **cmap**: [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

- **levels**: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass levels=[0]

- **origin**: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of \( Z \) will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.
  
  This keyword is not active if \( X \) and \( Y \) are specified in the call to contour.

- **extent**: [None | (x0,x1,y0,y1)]
  
  If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries.
  
  In this case, the position of \( Z[0,0] \) is the center of the pixel, not a corner.
If \textit{origin} is None, then \((x0, y0)\) is the position of \(Z[0,0]\), and \((xI, yI)\) is the position of \(Z[-1,-1]\).

This keyword is not active if \(X\) and \(Y\) are specified in the call to \texttt{contour}.

\textit{locator}: [ \texttt{None} | \texttt{ticker.Locator subclass} ] If \textit{locator} is None, the default \texttt{MaxNLocator} is used. The locator is used to determine the contour levels if they are not given explicitly via the \textit{V} argument.

\textit{extend}: [ \texttt{‘neither’} | \texttt{‘both’} | \texttt{‘min’} | \texttt{‘max’} ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via \texttt{matplotlib.colors.Colormap.set_under()} and \texttt{matplotlib.colors.Colormap.set_over()} methods.

\textit{xunits}, \textit{yunits}: [ \texttt{None} | \texttt{registered units} ] Override axis units by specifying an instance of a \texttt{matplotlib.units.ConversionInterface}.

\texttt{tricontour}-only keyword arguments:

\textit{linewidths}: [ \texttt{None} | \texttt{number} | \texttt{tuple of numbers} ] If \textit{linewidths} is None, the default width in \texttt{lines.linewidth} in \texttt{matplotlibrc} is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

\textit{linestyles}: [ \texttt{None} | \texttt{‘solid’} | \texttt{‘dashed’} | \texttt{‘dashdot’} | \texttt{‘dotted’} ] If \textit{linestyles} is None, the ‘solid’ is used.

\textit{linestyles} can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in \texttt{contour.negative_linestyle} in \texttt{matplotlibrc} will be used.

\texttt{tricontourf}-only keyword arguments:

\textit{antialiased}: [ \texttt{True} | \texttt{False} ] enable antialiasing

\textit{nchunk}: [ \texttt{0} | \texttt{integer} ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly \textit{nchunk} by \textit{nchunk} points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless \textit{antialiased} is False.

Note: \textit{tricontourf} fills intervals that are closed at the top; that is, for boundaries \(z1\) and \(z2\), the filled region is:

\[ z1 < z \leq z2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the \(z\) array, then that minimum value will be included in the lowest interval.

\textbf{Examples:}
Contour plot of Delaunay triangulation
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.tripcolor(*args, **kwargs)
```

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```
tripcolor(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tripcolor(x, y, ...)
tripcolor(x, y, triangles, ...)
tripcolor(x, y, triangles=triangles, ...)
tripcolor(x, y, mask=mask, ...)
tripcolor(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The next argument must be `C`, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the kwarg `facecolors` *= C instead of just *C.
shading may be ‘flat’ (the default) or ‘gouraud’. If shading is ‘flat’ and C values are defined at points, the color values used for each triangle are from the mean C of the triangle’s three points. If shading is ‘gouraud’ then color values must be defined at points. shading of ‘faceted’ is deprecated; please use edgecolors instead.

The remaining kwargs are the same as for pcolor().

Example:
Tripcolor of Delaunay triangulation, gouraud shading

Tripcolor of user-specified triangulation
Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.triplot(*args, **kwargs)
```

Draw a unstructured triangular grid as lines and/or markers.

The triangulation to plot can be specified in one of two ways; either:

```python
triplot(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```python
triplot(x, y, ...)  
triplot(x, y, triangles, ...)  
triplot(x, y, triangles=triangles, ...)  
triplot(x, y, mask=mask, ...)  
triplot(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining args and kwargs are the same as for `plot()`.

Return a list of 2 `Line2D` containing respectively:

- the lines plotted for triangles edges
- the markers plotted for triangles nodes

Example:

![Triplot of Delaunay triangulation](image)
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.twinx(ax=None)
```

Make a second axes that shares the x-axis. The new axes will overlay ax (or the current axes if ax is None). The ticks for ax2 will be placed on the right, and the ax2 instance is returned.

**See also:**

```
examples/api_examples/two_scales.py For an example
```

```
matplotlib.pyplot.twiny(ax=None)
```

Make a second axes that shares the y-axis. The new axis will overlay ax (or the current axes if ax is None). The ticks for ax2 will be placed on the top, and the ax2 instance is returned.

```
matplotlib.pyplot.violinplot(dataset, positions=None, vert=True, widths=0.5,
                            showmeans=False, showextrema=True, showmedians=False,
                            points=100, bw_method=None, hold=None)
```

Make a violin plot.

Call signature:

```
violinplot(dataset, positions=None, vert=True, widths=0.5,
            showmeans=False, showextrema=True, showmedians=False,
            points=100, bw_method=None):
```

Make a violin plot for each column of dataset or each vector in sequence dataset. Each filled area extends to represent the entire data range, with optional lines at the mean, the
median, the minimum, and the maximum.

**Parameters**

- **dataset**: Array or a sequence of vectors.
  The input data.
- **positions** [array-like, default = [1, 2, ..., n]] Sets the positions of the violins. The ticks and limits are automatically set to match the positions.
- **vert** [bool, default = True.] If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.
- **widths** [array-like, default = 0.5] Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.
- **showmeans** [bool, default = False] If True, will toggle rendering of the means.
- **showextrema** [bool, default = True] If True, will toggle rendering of the extrema.
- **showmedians** [bool, default = False] If True, will toggle rendering of the medians.
- **points** [scalar, default = 100] Defines the number of points to evaluate each of the gaussian kernel density estimations at.
- **bw_method** [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be ‘scott’, ‘silverman’, a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a `GaussianKDE` instance as its only parameter and return a scalar. If None (default), ‘scott’ is used.

**Returns**

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- **bodies**: A list of the `matplotlib.collections.PolyCollection` instances containing the filled area of each violin.
- **means**: A `matplotlib.collections.LineCollection` instance created to identify the mean values of each of the violin’s distribution.
- **mins**: A `matplotlib.collections.LineCollection` instance created to identify the bottom of each violin’s distribution.
- **maxes**: A `matplotlib.collections.LineCollection` instance created to identify the top of each violin’s distribution.
- **bars**: A `matplotlib.collections.LineCollection` instance created to identify the centers of each violin’s distribution.
- **medians**: A `matplotlib.collections.LineCollection` instance created to identify the median values of each of the violin’s distribution.

Additional kwargs: hold = [True|False] overrides default hold state
Plot vertical lines.

**Parameters**
- `x`: scalar or 1D array_like
  - x-indexes where to plot the lines.
- `ymin, ymax`: scalar or 1D array_like
  - Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- `colors`: array_like of colors, optional, default: ‘k’
- `linestyles`: [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’], optional
- `label`: string, optional, default: ‘’

**Returns**
- `lines`: LineCollection

**Other Parameters**
- `kwargs`: LineCollection properties.

**See also:**
- `hlines` horizontal lines

**Examples**

```
Additional kwargs: hold = [True|False] overrides default hold state
```

```python
matplotlib.pyplot.waitforbuttonpress(*args, **kwargs)
```

Call signature:

```
waitforbuttonpress(self, timeout=-1)
```

Blocking call to interact with the figure.

This will return True is a key was pressed, False if a mouse button was pressed and None if `timeout` was reached without either being pressed.
If `timeout` is negative, does not timeout.

```
matplotlib.pyplot.winter()
```

set the default colormap to winter and apply to current image if any. See help(colormaps) for more information

```
matplotlib.pyplot.xcorr(x, y, normed=True, detrend=<function detrend_none at 0x7f8bf872f320>, usevlines=True, maxlags=10, hold=None, **kwargs)
```

Plot the cross correlation between `x` and `y`.

**Parameters**
- `x`: sequence of scalars of length `n`
- `y`: sequence of scalars of length `n`
- `hold`: boolean, optional, default: True
- `detrend`: callable, optional, default: `mlab.detrend_none`
  - `x` is detrended by the `detrend` callable. Default is no normalization.
- `normed`: boolean, optional, default: True
  - if True, normalize the data by the autocorrelation at the 0-th lag.
- `usevlines`: boolean, optional, default: True
  - if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.
- `maxlags`: integer, optional, default: 10
  - number of lags to show. If None, will return all `2 * len(x) - 1` lags.

**Returns**
- `(lags, c, line, b)`: where:
  - `lags` are a length `2^maxlags+1` lag vector.
  - `c` is the `2^maxlags+1` auto correlation vector
  - `line` is a `Line2D` instance returned by `plot`.
  - `b` is the x-axis (none, if plot is used).

**Other Parameters**
- `linestyle`: `Line2D` prop, optional, default: None
  - Only used if usevlines is False.
- `marker`: string, optional, default: ‘o’

**Notes**

The cross correlation is performed with `numpy.correlate()` with mode = 2.

Additional kwargs: `hold = [True|False]` overrides default hold state

```
matplotlib.pyplot.xkcd(scale=1, length=100, randomness=2)
```

Turns on `xkcd` sketch-style drawing mode. This will only have effect on things drawn after this function is called.

For best results, the “Humor Sans” font should be installed: it is not included with matplotlib.

**Parameters**
- `scale`: float, optional
  - The amplitude of the wiggle perpendicular to the source line.
- `length`: float, optional
  - The length of the wiggle along the line.
- `randomness`: float, optional
  - The scale factor by which the length is shrunk or expanded.
Notes

This function works by a number of rcParams, so it will probably override others you have set before.

If you want the effects of this function to be temporary, it can be used as a context manager, for example:

```python
with plt.xkcd():
    # This figure will be in XKCD-style
    fig1 = plt.figure()
    # ...

    # This figure will be in regular style
    fig2 = plt.figure()
```

**matplotlib.pyplot.xlabel(s, *args, **kwargs)**

Set the x axis label of the current axis.

Default override is:

```python
override = {
    'fontsize': 'small',
    'verticalalignment': 'top',
    'horizontalalignment': 'center'
}
```

See also:

**text()** For information on how override and the optional args work.

**matplotlib.pyplot.xlim(*args, **kwargs)**

Get or set the x limits of the current axes.

```python
xmin, xmax = xlim()  # return the current xlim
xlim( (xmin, xmax) )  # set the xlim to xmin, xmax
xlim( xmin, xmax )    # set the xlim to xmin, xmax
```

If you do not specify args, you can pass the xmin and xmax as kwargs, e.g.:

```python
xlim(xmax=3)  # adjust the max leaving min unchanged
xlim(xmin=1)  # adjust the min leaving max unchanged
```

Setting limits turns autoscaling off for the x-axis.

The new axis limits are returned as a length 2 tuple.

**matplotlib.pyplot.xscale(*args, **kwargs)**

Set the scaling of the x-axis.

call signature:

```python
xscale(scale, **kwargs)
```
The available scales are: u'linear' | u'log' | u'symlog'

Different keywords may be accepted, depending on the scale:

‘linear’

‘log’

*basex/basey*: The base of the logarithm

*nonposx/nonposy*: [‘mask’ | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

*subsx/subsy*: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘symlog’

*basex/basey*: The base of the logarithm

*linthreshx/linthreshy*: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

*subsx/subsy*: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

*linscalex/linscaley*: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when linscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```
import matplotlib.pyplot as plt

# Get or set the x-limits of the current tick locations and labels.
locs, labels = plt.xticks()

plt.xticks(plt.arange(6))

plt.xticks(plt.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'))

plt.xticks(plt.arange(12), plt.calendar.month_name[1:13], rotation=17)

plt.ylabel('s', **kwargs)
```

Set the y axis label of the current axis.
Defaults override is:

```python
override = {
    'fontsize': 'small',
    'verticalalignment': 'center',
    'horizontalalignment': 'right',
    'rotation': 'vertical'
}
```

See also:

- `text()` For information on how override and the optional args work.

```python
matplotlib.pyplot.ylim(*args, **kwargs)
```

Get or set the y-limits of the current axes.

```python
ymin, ymax = ylim()  # return the current ylim
ylim( (ymin, ymax) )  # set the ylim to ymin, ymax
ylim( ymin, ymax )  # set the ylim to ymin, ymax
```

If you do not specify args, you can pass the `ymin` and `ymax` as kwargs, e.g.:

```python
ylim(ymax=3)  # adjust the max leaving min unchanged
ylim(ymin=1)  # adjust the min leaving max unchanged
```

Setting limits turns autoscaling off for the y-axis.

The new axis limits are returned as a length 2 tuple.

```python
matplotlib.pyplot.yscale(*args, **kwargs)
```

Set the scaling of the y-axis.

Call signature:

```python
yscale(scale, **kwargs)
```

The available scales are: `u'linear' | u'log' | u'symlog`

Different keywords may be accepted, depending on the scale:

- **linear**
- **log**
  - `basex/basey`: The base of the logarithm
  - `nonposx/nonposy`: `[mask] | `clip`] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
  - `subsx/subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`
    will place 8 logarithmically spaced minor ticks between each major tick.

- **symlog**
  - `basex/basey`: The base of the logarithm
**linthreshx/linthreshy**: The range \((-x, x)\) within which the plot is linear (to avoid having the plot go to infinity around zero).

**subsx/subsy**: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: \([2, 3, 4, 5, 6, 7, 8, 9]\) will place 8 logarithmically spaced minor ticks between each major tick.

**linscalex/linscaley**: This allows the linear range \((-\text{linthresh} \text{ to linthresh})\) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when \(\text{linscale} == 1.0\) (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```
matplotlib.pyplot.yticks(*args, **kwargs)
Get or set the y-limits of the current tick locations and labels.

# return locs, labels where locs is an array of tick locations and
# labels is an array of tick labels.
locs, labels = yticks()

# set the locations of the yticks
yticks( arange(6) )

# set the locations and labels of the yticks
yticks( arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue') )
```

The keyword args, if any, are Text properties. For example, to rotate long labels:

```
yticks( arange(12), calendar.month_name[1:13], rotation=45 )
```
64.1 `matplotlib.sankey`

Module for creating Sankey diagrams using matplotlib

```python
class matplotlib.sankey.Sankey(ax=None, scale=1.0, unit=u'', format=u'%.G', gap=0.25, radius=0.1, shoulder=0.03, offset=0.15, head_angle=100, margin=0.4, tolerance=1e-06, **kwargs)
```

Sankey diagram in matplotlib

Sankey diagrams are a specific type of flow diagram, in which the width of the arrows is shown proportionally to the flow quantity. They are typically used to visualize energy or material or cost transfers between processes. [Wikipedia (6/1/2011)](https://en.wikipedia.org/wiki/Sankey_diagram)

Create a new Sankey instance.

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ax</td>
<td>axes onto which the data should be plotted If ax isn’t provided, new axes will be created.</td>
</tr>
<tr>
<td>scale</td>
<td>scaling factor for the flows scale sizes the width of the paths in order to maintain proper layout. The same scale is applied to all subdiagrams. The value should be chosen such that the product of the scale and the sum of the inputs is approximately 1.0 (and the product of the scale and the sum of the outputs is approximately -1.0).</td>
</tr>
<tr>
<td>unit</td>
<td>string representing the physical unit associated with the flow quantities If unit is None, then none of the quantities are labeled.</td>
</tr>
<tr>
<td>format</td>
<td>a Python number formatting string to be used in labeling the flow as a quantity (i.e., a number times a unit, where the unit is given)</td>
</tr>
<tr>
<td>gap</td>
<td>space between paths that break in/break away to/from the top or bottom</td>
</tr>
<tr>
<td>radius</td>
<td>inner radius of the vertical paths</td>
</tr>
<tr>
<td>shoulder</td>
<td>size of the shoulders of output arrowS</td>
</tr>
<tr>
<td>offset</td>
<td>text offset (from the dip or tip of the arrow)</td>
</tr>
<tr>
<td>head_angle</td>
<td>angle of the arrow heads (and negative of the angle of the tails) [deg]</td>
</tr>
<tr>
<td>margin</td>
<td>minimum space between Sankey outlines and the edge of the plot area</td>
</tr>
<tr>
<td>tolerance</td>
<td>acceptable maximum of the magnitude of the sum of flows The magnitude of the sum of connected flows cannot be greater than tolerance.</td>
</tr>
</tbody>
</table>

The optional arguments listed above are applied to all subdiagrams so that there is consistent alignment and formatting.

If `Sankey` is instantiated with any keyword arguments other than those explicitly listed above (**kwargs), they will be passed to `add()`, which will create the first subdiagram.

In order to draw a complex Sankey diagram, create an instance of `Sankey` by calling it without any kwargs:

```python
sankey = Sankey()
```

Then add simple Sankey sub-diagrams:

```python
sankey.add() # 1
sankey.add() # 2
#. ...
sankey.add() # n
```

Finally, create the full diagram:

```python
sankey.finish()
```

Or, instead, simply daisy-chain those calls:

```python
Sankey().add().add... .add().finish()
```
See also:
add() finish()

Examples:
The default settings produce a diagram like this.
Flow Diagram of a Widget

Two Systems

Diagram showing the flow of a widget with percentages and categories. The diagram is labeled as 'Widget A' and includes sections for 'First', 'Second', 'Third', 'Fourth', and 'Fifth' with respective percentages. The diagram also shows a 'Hurray!' section with a 40% label.

Another diagram labeled 'Two Systems' shows a complex flowchart representing two systems, 'one' and 'two', represented by different colors and arrow directions.
add(patchlabel='u'', flows=None, orientations=None, labels='u''', trunklength=1.0, pathlengths=0.25, prior=None, connect=(0, 0), rotation=0, **kwargs)

Add a simple Sankey diagram with flows at the same hierarchical level.

Return value is the instance of Sankey.

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patchlabel</td>
<td>label to be placed at the center of the diagram Note: label (not patchlabel) will be passed to the patch through **kwargs and can be used to create an entry in the legend.</td>
</tr>
<tr>
<td>flows</td>
<td>array of flow values By convention, inputs are positive and outputs are negative.</td>
</tr>
<tr>
<td>orientations</td>
<td>list of orientations of the paths Valid values are 1 (from/to the top), 0 (from/to the left or right), or -1 (from/to the bottom). If orientations == 0, inputs will break in from the left and outputs will break away to the right.</td>
</tr>
<tr>
<td>labels</td>
<td>list of specifications of the labels for the flows Each value may be None (no labels), '' (just label the quantities), or a labeling string. If a single value is provided, it will be applied to all flows. If an entry is a non-empty string, then the quantity for the corresponding flow will be shown below the string. However, if the unit of the main diagram is None, then quantities are never shown, regardless of the value of this argument.</td>
</tr>
<tr>
<td>trunklength</td>
<td>length between the bases of the input and output groups</td>
</tr>
<tr>
<td>pathlengths</td>
<td>list of lengths of the arrows before break-in or after break-away If a single value is given, then it will be applied to the first (inside) paths on the top and bottom, and the length of all other arrows will be justified accordingly. The pathlengths are not applied to the horizontal inputs and outputs.</td>
</tr>
<tr>
<td>prior</td>
<td>index of the prior diagram to which this diagram should be connected a (prior, this) tuple indexing the flow of the prior diagram and the flow of this diagram which should be connected If this is the first diagram or prior is None, connect will be ignored.</td>
</tr>
<tr>
<td>connect</td>
<td>angle of rotation of the diagram [deg] rotation is ignored if this diagram is connected to an existing one (using prior and connect). The interpretation of the orientations argument will be rotated accordingly (e.g., if rotation == 90, an orientations entry of 1 means to/from the left).</td>
</tr>
<tr>
<td>rotation</td>
<td>angle of rotation of the diagram [deg] rotation is ignored if this diagram is connected to an existing one (using prior and connect). The interpretation of the orientations argument will be rotated accordingly (e.g., if rotation == 90, an orientations entry of 1 means to/from the left).</td>
</tr>
</tbody>
</table>

Valid kwargs are matplotlib.patches.PathPatch() arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 64.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

As examples, fill=False and label='A legend entry'. By default, facecolor='#bfd1d4’ (light blue) and linewidth=0.5.

The indexing parameters (prior and connect) are zero-based.

The flows are placed along the top of the diagram from the inside out in order of their index within the flows list or array. They are placed along the sides of the diagram from the top down and along the bottom from the outside in.

If the the sum of the inputs and outputs is nonzero, the discrepancy will appear as a cubic Bezier curve along the top and bottom edges of the trunk.

**See also:**

finish()

finish()

Adjust the axes and return a list of information about the Sankey subdiagram(s).
Return value is a list of subdiagrams represented with the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patch</td>
<td>Sankey outline (an instance of PathPatch)</td>
</tr>
<tr>
<td>flows</td>
<td>values of the flows (positive for input, negative for output)</td>
</tr>
<tr>
<td>angles</td>
<td>list of angles of the arrows [\text{deg}/90] For example, if the diagram has not been rotated, an input to the top side will have an angle of 3 (DOWN), and an output from the top side will have an angle of 1 (UP). If a flow has been skipped (because its magnitude is less than tolerance), then its angle will be None.</td>
</tr>
<tr>
<td>tips</td>
<td>array in which each row is an ([x, y]) pair indicating the positions of the tips (or “dips”) of the flow paths If the magnitude of a flow is less the tolerance for the instance of Sankey, the flow is skipped and its tip will be at the center of the diagram.</td>
</tr>
<tr>
<td>text</td>
<td>Text instance for the label of the diagram</td>
</tr>
<tr>
<td>texts</td>
<td>list of Text instances for the labels of flows</td>
</tr>
</tbody>
</table>

See also:

add()
65.1 matplotlib.spines

class matplotlib.spines.Spine(axes, spine_type, path, **kwargs)

an axis spine – the line noting the data area boundaries

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions. See function: set_position for more information.

The default position is ('outward', 0).

Spines are subclasses of class: Patch, and inherit much of their behavior.

Spines draw a line or a circle, depending if function: set_patch_line or function: set_patch_circle has been called. Line-like is the default.

• axes: the Axes instance containing the spine
• spine_type: a string specifying the spine type
• path: the path instance used to draw the spine

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 65.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/&quot;</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

classmethod `circular_spine`(axes, center, radius, **kwargs)

(staticmethod) Returns a circular Spine.

cla()

Clear the current spine
draw(artist, renderer, *args, **kwargs)

get_bounds()

Get the bounds of the spine.
get_patch_transform()

gget_path()

gget_position()

get the spine position
gget_smart_bounds()

get whether the spine has smart bounds
gget_spine_transform()

get the spine transform

is_frame_like()

return True if directly on axes frame

This is useful for determining if a spine is the edge of an old style MPL plot. If so, this function will return True.
**classmethod** `linear_spine`(*axes, spine_type*, **kwargs)

Returns a linear `Spine`.

**register_axis** (*axis*)

register an axis

An axis should be registered with its corresponding spine from the Axes instance. This allows the spine to clear any axis properties when needed.

**set_bounds** (*low, high*)

Set the bounds of the spine.

**set_color** (*c*)

Set the edgecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

**set_patch_circle** (*center, radius*)

set the spine to be circular

**set_patch_line**()

set the spine to be linear

**set_position** (*position*)

set the position of the spine

Spine position is specified by a 2 tuple of (position type, amount). The position types are:

- 'outward' : place the spine out from the data area by the specified number of points.
  (Negative values specify placing the spine inward.)
- 'axes' : place the spine at the specified Axes coordinate (from 0.0-1.0).
- 'data' : place the spine at the specified data coordinate.

Additionally, shorthand notations define a special positions:

- 'center' -> ('axes', 0.5)
- 'zero' -> ('data', 0.0)

**set_smart_bounds** (*value*)

set the spine and associated axis to have smart bounds
66.1 matplotlib.style

matplotlib.style.context(*args, **kwds)

Context manager for using style settings temporarily.

Parameters

- name : str or list of str
  Name of style or path/URL to a style file. For a list of available style names, see style.available. If given a list, each style is applied from first to last in the list.

- after_reset : bool
  If True, apply style after resetting settings to their defaults; otherwise, apply style on top of the current settings.

matplotlib.style.reload_library()

Reload style library.

matplotlib.style.use(name)

Use matplotlib style settings from a known style sheet or from a file.

Parameters

- name : str or list of str
  Name of style or path/URL to a style file. For a list of available style names, see style.available. If given a list, each style is applied from first to last in the list.

matplotlib.style.library

Dictionary of available styles

matplotlib.style.available

List of available styles
67.1 matplotlib.text

Classes for including text in a figure.

```python
class matplotlib.text.Annotation(s, xy, xytext=None, xycoords=u'data', textcoords=None, arrowprops=None, annotation_clip=None, **kwargs)
```

Bases: `matplotlib.text.Text`, `matplotlib.text._AnnotationBase`

A Text class to make annotating things in the figure, such as Figure, Axes, Rectangle, etc., easier.

Annotate the $x$, $y$ point $xy$ with text $s$ at $x$, $y$ location $xytext$. (If $xytext = None$, defaults to $xy$, and if $textcoords = None$, defaults to $xycoords$).

`arrowprops`, if not `None`, is a dictionary of line properties (see `matplotlib.lines.Line2D`) for the arrow that connects annotation to the point.

If the dictionary has a key `arrowstyle`, a `FancyArrowPatch` instance is created with the given dictionary and is drawn. Otherwise, a `YAArrow` patch instance is created and drawn. Valid keys for `YAArrow` are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>oftentimes it is convenient to have the arrowtip and base a bit away from the text and point being annotated. If $d$ is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shrink percent of the distance $d$ away from the endpoints. i.e., $\text{shrink=0.05}$ is 5%</td>
</tr>
<tr>
<td>?</td>
<td>any key for <code>matplotlib.patches.polygon</code></td>
</tr>
</tbody>
</table>

Valid keys for `FancyArrowPatch` are:
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
<tr>
<td>?</td>
<td>any key for <code>matplotlib.patches.PathPatch</code></td>
</tr>
</tbody>
</table>

`xycoords` and `textcoords` are strings that indicate the coordinates of `xy` and `xytext`, and may be one of the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'figure points'</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure pixels'</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure fraction'</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>'axes points'</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>'axes pixels'</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>'axes fraction'</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>'data'</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>'offset points'</td>
<td>Specify an offset (in points) from the xy value</td>
</tr>
<tr>
<td>'polar'</td>
<td>you can specify <code>theta</code>, <code>r</code> for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native “data” coordinate system.</td>
</tr>
</tbody>
</table>

If a ‘points’ or ‘pixels’ option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. e.g.:

```python
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of `Transform` or `Artist`. See `Annotating Axes` for more details.

The `annotation_clip` attribute controls the visibility of the annotation when it goes outside the axes area. If `True`, the annotation will only be drawn when the `xy` is inside the axes. If `False`, the
annotation will always be drawn regardless of its position. The default is None, which behave as True only if xycoords is “data”.

Additional kwarg are Text properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>bgcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean]callable]</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>text</td>
<td>Transform instance</td>
</tr>
<tr>
<td>transform</td>
<td>a url string</td>
</tr>
<tr>
<td>url</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
anncoords

contains(event)

draw(artist, renderer, *args, **kwargs)
   Draw the Annotation object to the given renderer.

get_window_extent(renderer=None)
   Return a Bbox object bounding the text and arrow annotation, in display units.

   renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of draw(), so you must use this kwarg if you want to call get_window_extent() prior to the first draw(). For getting web page regions, it is simpler to call the method after saving the figure. The dpi used defaults to self.figure.dpi; the renderer dpi is irrelevant.

set_figure(fig)

update_bbox_position_size(renderer)
   Update the location and the size of the bbox. This method should be used when the position and size of the bbox needs to be updated before actually drawing the bbox.

update_positions(renderer)
   "Update the pixel positions of the annotated point and the text.

xyann
class matplotlib.text.OffsetFrom(artist, ref_coord, unit='points')
   Bases: object

   get_unit()

   set_unit(unit)

class matplotlib.text.Text(x=0, y=0, text='', color=None, verticalalignment='baseline',
   horizontalalignment='left', multialignment=None, font-properties=None, rotation=None, linespacing=None, rotation_mode=None, **kwargs)
   Bases: matplotlib.artist.Artist

   Handle storing and drawing of text in window or data coordinates.

   Create a Text instance at x, y with string text.

   Valid kwargs are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>bgcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>['center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>['normal'</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>['normal'</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>['center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**contains**(mouseevent)

Test whether the mouse event occurred in the patch.

In the case of text, a hit is true anywhere in the axis-aligned bounding-box containing the text.
Returns True or False.

def draw(artist, renderer, *args, **kwargs)
    Draws the Text object to the given renderer.

def get_bbox_patch()
    Return the bbox Patch object. Returns None if the the FancyBboxPatch is not made.

def get_color()
    Return the color of the text

def get_family()
    Return the list of font families used for font lookup

def get_font_properties()
    alias for get_fontproperties

def get_fontfamily()
    alias for get_family

def get_fontname()
    alias for get_name

def get_fontproperties()
    Return the FontProperties object

def get_fontsize()
    alias for get_size

def get_fontstretch()
    alias for get_stretch

def get_fontstyle()
    alias for get_style

def get_fontvariant()
    alias for get_variant

def get_fontweight()
    alias for get_weight

def get_ha()
    alias for get_horizontalalignment

def get_horizontalalignment()
    Return the horizontal alignment as string. Will be one of ‘left’, ‘center’ or ‘right’.

def get_name()
    Return the font name as string

def get_position()
    Return the position of the text as a tuple (x, y)

def get_prop_tup()
    Return a hashable tuple of properties.
Not intended to be human readable, but useful for backends who want to cache derived information about text (e.g., layouts) and need to know if the text has changed.

```python
def get_rotation()
    return the text angle as float in degrees

def get_rotation_mode()
    get text rotation mode

def get_size()
    Return the font size as integer

def get_stretch()
    Get the font stretch as a string or number

def get_style()
    Return the font style as string

def get_text()
    Get the text as string

def get_va()
    alias for getverticalalignment()

def get_variant()
    Return the font variant as a string

def get_verticalalignment()
    Return the vertical alignment as string. Will be one of ‘top’, ‘center’, ‘bottom’ or ‘baseline’.

def get_weight()
    Get the font weight as string or number

def get_window_extent(renderer=None, dpi=None)
    Return a Bbox object bounding the text, in display units.

    In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

    renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of draw(), so you must use this kwarg if you want to call get_window_extent() prior to the first draw(). For getting web page regions, it is simpler to call the method after saving the figure.

    dpi defaults to self.figure.dpi; the renderer dpi is irrelevant. For the web application, if figure.dpi is not the value used when saving the figure, then the value that was used must be specified as the dpi argument.

static is_math_text(s)
    Returns a cleaned string and a boolean flag. The flag indicates if the given string s contains any mathtext, determined by counting unescaped dollar signs. If no mathtext is present, the cleaned string has its dollar signs unescaped. If usetex is on, the flag always has the value “TeX”.

    Set the background color of the text by updating the bbox.
```
See also:

`set_bbox()` To change the position of the bounding box.

ACCEPTS: any matplotlib color

`set_bbox(rectprops)`
Draw a bounding box around self. rectprops are any settable properties for a rectangle, e.g.,
facecolor='red', alpha=0.5.

t.set_bbox(dict(facecolor='red', alpha=0.5))
If rectprops has “boxstyle” key. A FancyBboxPatch is initialized with rectprops and will be
drawn. The mutation scale of the FancyBboxPath is set to the fontsize.

ACCEPTS: rectangle prop dict

`set_clip_box(clipbox)`
Set the artist’s clip Bbox.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

`set_clip_on(b)`
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

`set_clip_path(path, transform=None)`
Set the artist’s clip path, which may be:
• a Patch (or subclass) instance
• a Path instance, in which case an optional Transform instance may be provided,
  which will be applied to the path before using it for clipping.
• None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the
clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [(Path, Transform)| Patch | None ]

`set_color(color)`
Set the foreground color of the text

ACCEPTS: any matplotlib color

`set_family(fontname)`
Set the font family. May be either a single string, or a list of strings in decreasing priority. Each
string may be either a real font name or a generic font class name. If the latter, the specific font
names will be looked up in the matplotlibrc file.

ACCEPTS: [FONTNAME | ‘serif’ | ‘sans-serif’ | ‘cursive’ | ‘fantasy’ | ‘monospace’ ]

`set_font_properties(fp)`
alias for set_fontproperties

`set_fontname(fontname)`
alias for set_family
set_fontproperties(fp)
Set the font properties that control the text. fp must be a matplotlib.font_manager.FontProperties object.

ACCEPTS: a matplotlib.font_manager.FontProperties instance

set_fontsize(fontsize)
alias for set_size

set_fontstretch(stretch)
alias for set_stretch

set_fontstyle(fontstyle)
alias for set_style

set_fontvariant(variant)
alias for set_variant

set_fontweight(weight)
alias for set_weight

set_ha(align)
alias for set_horizontalalignment

set_horizontalalignment(align)
Set the horizontal alignment to one of

ACCEPTS: [ ‘center’ | ‘right’ | ‘left’ ]

set_linespacing(spacing)
Set the line spacing as a multiple of the font size. Default is 1.2.

ACCEPTS: float (multiple of font size)

set_ma(align)
alias for set_verticalalignment

set_multialignment(align)
Set the alignment for multiple lines layout. The layout of the bounding box of all the lines is determined by the horizontalalignment and verticalalignment properties, but the multiline text within that box can be

ACCEPTS: [‘left’ | ‘right’ | ‘center’ ]

set_name(fontname)
alias for set_family

set_position(xy)
Set the (x, y) position of the text

ACCEPTS: (x,y)

set_rotation(s)
Set the rotation of the text

ACCEPTS: [ angle in degrees | ‘vertical’ | ‘horizontal’ ]
**set_rotation_mode**\( (m) \)
set text rotation mode. If “anchor”, the un-rotated text will first aligned according to their \texttt{ha} and \texttt{va}, and then will be rotated with the alignment reference point as a origin. If None (default), the text will be rotated first then will be aligned.

**set_size**\( (\text{fontsize}) \)
Set the font size. May be either a size string, relative to the default font size, or an absolute font size in points.

\textbf{ACCEPTS:} [size in points | ‘xx-small’ | ‘x-small’ | ‘small’ | ‘medium’ | ‘large’ | ‘x-large’ | ‘xx-large’ ]

**set_stretch**\( (\text{stretch}) \)
Set the font stretch (horizontal condensation or expansion).


**set_style**\( (\text{fontstyle}) \)
Set the font style.

\textbf{ACCEPTS:} [ ‘normal’ | ‘italic’ | ‘oblique’ ]

**set_text**\( (s) \)
Set the text string \texttt{s}

It may contain newlines (\texttt{\textbackslash n}) or math in LaTeX syntax.

\textbf{ACCEPTS:} string or anything printable with ‘%s’ conversion.

**set_va**\( (align) \)
alias for \texttt{set_verticalalignment}

**set_variant**\( (\text{variant}) \)
Set the font variant, either ‘normal’ or ‘small-caps’.

\textbf{ACCEPTS:} [ ‘normal’ | ‘small-caps’ ]

**set_verticalalignment**\( (align) \)
Set the vertical alignment

\textbf{ACCEPTS:} [ ‘center’ | ‘top’ | ‘bottom’ | ‘baseline’ ]

**set_weight**\( (\text{weight}) \)
Set the font weight.


**set_x**\( (x) \)
Set the \texttt{x} position of the text

\textbf{ACCEPTS:} float

**set_y**\( (y) \)
Set the \texttt{y} position of the text
ACCEPST: float

@update_bbox_position_size(renderer)
Update the location and the size of the bbox. This method should be used when the position and size of the bbox needs to be updated before actually drawing the bbox.

@update_from(other)
Copy properties from other to self

zorder = 3

class matplotlib.text.TextWithDash(x=0, y=0, text=u'', color=None, verticalalignment=u'center', horizontalalignment=u'center', multialignment=None, fontproperties=None, rotation=None, linespacing=None, dashlength=0.0, dashdirection=0, dashrotation=None, dashpad=3, dashpush=0)

Bases: matplotlib.text.Text

This is basically a Text with a dash (drawn with a Line2D) before/after it. It is intended to be a drop-in replacement for Text, and should behave identically to it when dashlength = 0.0.

The dash always comes between the point specified by set_position() and the text. When a dash exists, the text alignment arguments (horizontalalignment, verticalalignment) are ignored.

dashlength is the length of the dash in canvas units. (default = 0.0).
dashdirection is one of 0 or 1, where 0 draws the dash after the text and 1 before. (default = 0).
dashrotation specifies the rotation of the dash, and should generally stay None. In this case get_dashrotation() returns get_rotation(). (i.e., the dash takes its rotation from the text’s rotation). Because the text center is projected onto the dash, major deviations in the rotation cause what may be considered visually unappealing results. (default = None)
dashpad is a padding length to add (or subtract) space between the text and the dash, in canvas units. (default = 3)
dashpush “pushes” the dash and text away from the point specified by set_position() by the amount in canvas units. (default = 0)

Note: The alignment of the two objects is based on the bounding box of the Text, as obtained by get_window_extent(). This, in turn, appears to depend on the font metrics as given by the rendering backend. Hence the quality of the “centering” of the label text with respect to the dash varies depending on the backend used.

Note: I’m not sure that I got the get_window_extent() right, or whether that’s sufficient for providing the object bounding box.

draw(renderer)
Draw the TextWithDash object to the given renderer.

get_dashdirection()
Get the direction dash. 1 is before the text and 0 is after.
get_dashlength()
    Get the length of the dash.

get_dashpad()
    Get the extra spacing between the dash and the text, in canvas units.

get_dashpush()
    Get the extra spacing between the dash and the specified text position, in canvas units.

get_dashrotation()
    Get the rotation of the dash in degrees.

get_figure()
    return the figure instance the artist belongs to

get_position()
    Return the position of the text as a tuple (x, y)

get_prop_tup()
    Return a hashable tuple of properties.
    Not intended to be human readable, but useful for backends who want to cache derived information about text (e.g., layouts) and need to know if the text has changed.

get_window_extent(renderer=None)
    Return a Bbox object bounding the text, in display units.
    In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.
    
    renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of draw(), so you must use this kwarg if you want to call get_window_extent() prior to the first draw(). For getting web page regions, it is simpler to call the method after saving the figure.

set_dashdirection(dd)
    Set the direction of the dash following the text. 1 is before the text and 0 is after. The default is 0, which is what you’d want for the typical case of ticks below and on the left of the figure.
    
    ACCEPTS: int (1 is before, 0 is after)

set_dashlength(dl)
    Set the length of the dash.
    
    ACCEPTS: float (canvas units)

set_dashpad(dp)
    Set the “pad” of the TextWithDash, which is the extra spacing between the dash and the text, in canvas units.
    
    ACCEPTS: float (canvas units)

set_dashpush(dp)
    Set the “push” of the TextWithDash, which is the extra spacing between the beginning of the dash and the specified position.
    
    ACCEPTS: float (canvas units)
set_dashrotation(dr)
    Set the rotation of the dash, in degrees
    ACCEPTS: float (degrees)

set_figure(fig)
    Set the figure instance the artist belong to.
    ACCEPTS: a matplotlib.figure.Figure instance

set_position(xy)
    Set the (x, y) position of the TextWithDash.
    ACCEPTS: (x, y)

set_transform(t)
    Set the matplotlib.transforms.Transform instance used by this artist.
    ACCEPTS: a matplotlib.transforms.Transform instance

set_x(x)
    Set the x position of the TextWithDash.
    ACCEPTS: float

set_y(y)
    Set the y position of the TextWithDash.
    ACCEPTS: float

update_coords(renderer)
    Computes the actual x, y coordinates for text based on the input x, y and the dashlength. Since the rotation is with respect to the actual canvas’s coordinates we need to map back and forth.

matplotlib.text.get_rotation(rotation)
    Return the text angle as float.

    rotation may be ‘horizontal’, ‘vertical’, or a numeric value in degrees.
68.1 matplotlib.ticker

68.1.1 Tick locating and formatting

This module contains classes to support completely configurable tick locating and formatting. Although the locators know nothing about major or minor ticks, they are used by the Axis class to support major and minor tick locating and formatting. Generic tick locators and formatters are provided, as well as domain specific custom ones.

Default Formatter

The default formatter identifies when the x-data being plotted is a small range on top of a large offset. To reduce the chances that the ticklabels overlap the ticks are labeled as deltas from a fixed offset. For example:

```
ax.plot(np.arange(2000, 2010), range(10))
```

will have tick of 0-9 with an offset of +2e3. If this is not desired turn off the use of the offset on the default formatter:

```
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```

set the rcParam `axes.formatter.useoffset=False` to turn it off globally, or set a different formatter.

Tick locating

The Locator class is the base class for all tick locators. The locators handle autoscaling of the view limits based on the data limits, and the choosing of tick locations. A useful semi-automatic tick locator is MultipleLocator. You initialize this with a base, e.g., 10, and it picks axis limits and ticks that are multiples of your base.

The Locator subclasses defined here are

- **NullLocator**   No ticks
- **FixedLocator**  Tick locations are fixed
IndexLocator locator for index plots (e.g., where x = range(len(y)))
LinearLocator evenly spaced ticks from min to max
LogLocator logarithmically ticks from min to max
MultipleLocator

ticks and range are a multiple of base; either integer or float
OldAutoLocator choose a MultipleLocator and dynamically reassign it for intelligent ticking during navigation
MaxNLocator finds up to a max number of ticks at nice locations
AutoLocator MaxNLocator with simple defaults. This is the default tick locator for most plotting.
AutoMinorLocator locator for minor ticks when the axis is linear and the major ticks are uniformly spaced. It subdivides the major tick interval into a specified number of minor intervals, defaulting to 4 or 5 depending on the major interval.

There are a number of locators specialized for date locations - see the dates module

You can define your own locator by deriving from Locator. You must override the __call__ method, which returns a sequence of locations, and you will probably want to override the autoscale method to set the view limits from the data limits.

If you want to override the default locator, use one of the above or a custom locator and pass it to the x or y axis instance. The relevant methods are:

```python
ax.xaxis.set_major_locator( xmajorLocator )
ax.xaxis.set_minor_locator( xminorLocator )
ax.yaxis.set_major_locator( ymajorLocator )
ax.yaxis.set_minor_locator( yminorLocator )
```

The default minor locator is the NullLocator, e.g., no minor ticks on by default.

Tick formatting

Tick formatting is controlled by classes derived from Formatter. The formatter operates on a single tick value and returns a string to the axis.

NullFormatter no labels on the ticks
IndexFormatter set the strings from a list of labels
FixedFormatter set the strings manually for the labels
FuncFormatter user defined function sets the labels
FormatStrFormatter use a sprintf format string
ScalarFormatter default formatter for scalars; autopick the fmt string
LogFormatter formatter for log axes
You can derive your own formatter from the Formatter base class by simply overriding the \_\_call\_\_ method. The formatter class has access to the axis view and data limits.

To control the major and minor tick label formats, use one of the following methods:

ax.xaxis.set_major_formatter( xmajorFormatter )
ax.xaxis.set_minor_formatter( xminorFormatter )
ax.yaxis.set_major_formatter( ymajorFormatter )
ax.yaxis.set_minor_formatter( yminorFormatter )

See \_\_pylab_examples example code: major\_minor\_demo1.py for an example of setting major and minor
ticks. See the \_\_matplotlib.dates module for more information and examples of using date locators and
formatters.

class matplotlib.ticker.TickHelper
    Bases: object

    axis = None

    create_dummy_axis(**kwargs)

    set_axis(axis)

    set_bounds(vmin, vmax)

    set_data_interval(vmin, vmax)

    set_view_interval(vmin, vmax)

class matplotlib.ticker.Formatter
    Bases: matplotlib.ticker.TickHelper

    Convert the tick location to a string

    fix_minus(s)
        Some classes may want to replace a hyphen for minus with the proper unicode symbol
        (U+2212) for typographical correctness. The default is to not replace it.

        Note, if you use this method, e.g., in format_data() or call, you probably don’t want to use
        it for format_data_short() since the toolbar uses this for interactive coord reporting and I
        doubt we can expect GUIs across platforms will handle the unicode correctly. So for now the
        classes that override fix_minus() should have an explicit format_data_short() method

    format_data(value)

    format_data_short(value)
        return a short string version
get_offset()

locs = []

set_locs(locs)

class matplotlib.ticker.FixedFormatter(seq)
    Bases: matplotlib.ticker.Formatter
    Return fixed strings for tick labels
    seq is a sequence of strings. For positions i < len(seq) return seq[i] regardless of x. Otherwise return ‘’
    get_offset()

    set_offset_string(ofs)

class matplotlib.ticker.NullFormatter
    Bases: matplotlib.ticker.Formatter
    Always return the empty string

class matplotlib.ticker.FuncFormatter(func)
    Bases: matplotlib.ticker.Formatter
    User defined function for formatting

class matplotlib.ticker.FormatStrFormatter(fmt)
    Bases: matplotlib.ticker.Formatter
    Use an old-style (‘%’ operator) format string to format the tick

class matplotlib.ticker.ScalarFormatter(useOffset=None, useMathText=None, useLocale=None)
    Bases: matplotlib.ticker.Formatter
    Tick location is a plain old number. If useOffset==True and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. Scientific notation is used for data < 10^-n or data >= 10^m, where n and m are the power limits set using set_powerlimits((n,m)). The defaults for these are controlled by the axes.formatter.limits rc parameter.

    fix_minus(s)
        use a unicode minus rather than hyphen

    format_data(value)
        return a formatted string representation of a number

    format_data_short(value)
        return a short formatted string representation of a number
get_offset()
   Return scientific notation, plus offset

get_useLocale()

get_useOffset()

pprint_val(x)

set_locs(locs)
   set the locations of the ticks

set_powerlimits(lims)
   Sets size thresholds for scientific notation.
   e.g., formatter.set_powerlimits((-3, 4)) sets the pre-2007 default in which scientific
   notation is used for numbers less than 1e-3 or greater than 1e4. See also set_scientific().

set_scientific(b)
   True or False to turn scientific notation on or off see also set_powerlimits() 

set_useLocale(val)

set_useOffset(val)

useLocale

useOffset

class matplotlib.ticker.LogFormatter(base=10.0, labelOnlyBase=True)
   Bases: matplotlib.ticker.Formatter

Format values for log axis;
base is used to locate the decade tick, which will be the only one to be labeled if labelOnlyBase is
False

base(base)
   change the base for labeling - warning: should always match the base used for LogLocator

format_data(value)

format_data_short(value)
   return a short formatted string representation of a number

label_minor(labelOnlyBase)
   switch on/off minor ticks labeling
pprint_val(x, d)

class matplotlib.ticker.LogFormatterExponent(base=10.0, labelOnlyBase=True)
Bases: matplotlib.ticker.LogFormatter

Format values for log axis; using $exponent = \log\_base(value)$

$base$ is used to locate the decade tick, which will be the only one to be labeled if $labelOnlyBase$ is False

class matplotlib.ticker.LogFormatterMathtext(base=10.0, labelOnlyBase=True)
Bases: matplotlib.ticker.LogFormatter

Format values for log axis; using $exponent = \log\_base(value)$

$base$ is used to locate the decade tick, which will be the only one to be labeled if $labelOnlyBase$ is False

class matplotlib.ticker.Locator
Bases: matplotlib.ticker.TickHelper

Determine the tick locations;

Note, you should not use the same locator between different Axis because the locator stores references to the Axis data and view limits

MAXTICKS = 1000

autoscale()
autoscale the view limits

pan(numsteps)
Pan numticks (can be positive or negative)

raise_if_exceeds(locs)
raise a RuntimeError if Locator attempts to create more than MAXTICKS locs

refresh()
refresh internal information based on current lim

tick_values(vmin, vmax)
Return the values of the located ticks given vmin and vmax.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated axis simply call the Locator instance:

```python
>>> print((type(loc)))
<type 'Locator'>
>>> print((loc()))
[1, 2, 3, 4]
```

view_limits(vmin, vmax)
select a scale for the range from vmin to vmax
Normally this method is overridden by subclasses to change locator behaviour.

```
zoom(direction)
    Zoom in/out on axis; if direction is >0 zoom in, else zoom out
```

**class matplotlib.ticker.IndexLocator(base, offset)**

Bases: `matplotlib.ticker.Locator`

Place a tick on every multiple of some base number of points plotted, e.g., on every 5th point. It is assumed that you are doing index plotting; i.e., the axis is 0, len(data). This is mainly useful for x ticks.

```
place ticks on the i-th data points where (i-offset)%base==0
```

**tick_values(vmin, vmax)**

**class matplotlib.ticker.FixedLocator(locs, nbins=None)**

Bases: `matplotlib.ticker.Locator`

Tick locations are fixed. If nbins is not None, the array of possible positions will be subsampled to keep the number of ticks <= nbins +1. The subsampling will be done so as to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it is guaranteed to be one of the chosen ticks.

```
tick_values(vmin, vmax)
    " Return the locations of the ticks.
```

**Note:** Because the values are fixed, vmin and vmax are not used in this method.

**class matplotlib.ticker_NullLocator**

Bases: `matplotlib.ticker.Locator`

No ticks

```
tick_values(vmin, vmax)
    " Return the locations of the ticks.
```

**Note:** Because the values are Null, vmin and vmax are not used in this method.

**class matplotlib.ticker.LinearLocator(numticks=None, presets=None)**

Bases: `matplotlib.ticker.Locator`

Determine the tick locations

```
The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter the number of ticks will be fixed so that interactive navigation will be nice
Use presets to set locs based on lom. A dict mapping vmin, vmax->locs
```

```
tick_values(vmin, vmax)
```

```
view_limits(vmin, vmax)
    Try to choose the view limits intelligently
```
class matplotlib.ticker.LogLocator(
    base=10.0, subs=[1.0], numdec=4, numticks=15)
Bases: matplotlib.ticker.Locator

Determine the tick locations for log axes
place ticks on the location= base**i*subs[j]

base(base)
    set the base of the log scaling (major tick every base**i, i integer)

subs(subs)
    set the minor ticks the log scaling every base**i*subs[j]

tick_values(vmin, vmax)

view_limits(vmin, vmax)
    Try to choose the view limits intelligently

class matplotlib.ticker.AutoLocator
    Bases: matplotlib.ticker.MaxNLocator

class matplotlib.ticker.MultipleLocator(base=1.0)
    Bases: matplotlib.ticker.Locator

Set a tick on every integer that is multiple of base in the view interval

tick_values(vmin, vmax)

view_limits(dmin, dmax)
    Set the view limits to the nearest multiples of base that contain the data

class matplotlib.ticker.MaxNLocator(*args, **kwargs)
    Bases: matplotlib.ticker.Locator

Select no more than N intervals at nice locations.

Keyword args:
    nbins Maximum number of intervals; one less than max number of ticks.
    steps Sequence of nice numbers starting with 1 and ending with 10; e.g., [1, 2, 4, 5, 10]
    integer If True, ticks will take only integer values.
    symmetric If True, autoscaling will result in a range symmetric about zero.
    prune ['lower' | 'upper' | 'both' | None] Remove edge ticks – useful for stacked or ganged plots where
        the upper tick of one axes overlaps with the lower tick of the axes above it. If prune==’lower’,
        the smallest tick will be removed. If prune==’upper’, the largest tick will be removed. If
        prune==’both’, the largest and smallest ticks will be removed. If prune==None, no ticks will
        be removed.
    bin_boundaries(vmin, vmax)

default_params = {'trim': True, 'nbins': 10, 'steps': None, 'prune': None, 'integer': False, 'symmetric': False}

set_params(**kwargs)
```python

tick_values(vmin, vmax)

view_limits(dmin, dmax)

class matplotlib.ticker.AutoMinorLocator(n=None)
   Bases: matplotlib.ticker.Locator

   Dynamically find minor tick positions based on the positions of major ticks. Assumes the scale is linear and major ticks are evenly spaced.

   n is the number of subdivisions of the interval between major ticks; e.g., n=2 will place a single minor tick midway between major ticks.

   If n is omitted or None, it will be set to 5 or 4.

   tick_values(vmin, vmax)
```
69.1 matplotlib.tight_layout

This module provides routines to adjust subplot params so that subplots are nicely fit in the figure. In doing so, only axis labels, tick labels and axes titles are currently considered.

Internally, it assumes that the margins (left_margin, etc.) which are differences between ax.get_tightbbox and ax.bbox are independent of axes position. This may fail if Axes.adjustable is datalim. Also, This will fail for some cases (for example, left or right margin is affected by xlabel).

```python
matplotlib.tight_layout.auto_adjust_subplotpars(fig, renderer, nrows_ncols, num1num2_list, subplot_list, ax_bbox_list=None, pad=1.08, h_pad=None, w_pad=None, rect=None)
```

Return a dictionary of subplot parameters so that spacing between subplots are adjusted. Note that this function ignore geometry information of subplot itself, but uses what is given by nrows_ncols and num1num2_list parameteres. Also, the results could be incorrect if some subplots have adjustable=datalim.

Parameters:
- **nrows_ncols**: number of rows and number of columns of the grid.
- **num1num2_list**: list of numbers specifying the area occupied by the subplot
- **subplot_list**: list of subplots that will be used to calculate optimal subplot_params.
- **pad**: [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.
- **h_pad, w_pad**: [float] (padding (height/width) between edges of adjacent subplots). Defaults to pad_inches.
- **rect**: [left, bottom, right, top] in normalized (0, 1) figure coordinates.

```python
matplotlib.tight_layout.get_renderer(fig)
```

```python
matplotlib.tight_layout.get_subplotspec_list(axes_list, grid_spec=None)
```

Return a list of subplotspec from the given list of axes. For an instance of axes that does not support subplotspec, None is inserted in the list.

If grid_spec is given, None is inserted for those not from the given grid_spec.
matplotlib.tight_layout.get_tight_layout_figure(fig, axes_list, subplotspec_list, renderer, pad=1.08, h_pad=None, w_pad=None, rect=None)

Return subplot parameters for tight-layouted-figure with specified padding.

Parameters:

fig : figure instance

axes_list : a list of axes

subplotspec_list [a list of subplotspec associated with each] axes in axes_list

renderer : renderer instance

pad [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad, w_pad [float] padding (height/width) between edges of adjacent subplots. Defaults to pad_inches.

rect [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).
CHAPTER SEVENTY

TRIANGULAR GRIDS

70.1 matplotlib.tri

Unstructured triangular grid functions.

class matplotlib.tri.Triangulation(x, y, triangles=None, mask=None)

An unstructured triangular grid consisting of npoints points and ntri triangles. The triangles can either be specified by the user or automatically generated using a Delaunay triangulation.

Parameters:
- `x, y`: array_like of shape (npoints)
  Coordinates of grid points.
- `triangles`: integer array_like of shape (ntri, 3), optional
  For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.
- `mask`: boolean array_like of shape (ntri), optional
  Which triangles are masked out.

Notes

For a Triangulation to be valid it must not have duplicate points, triangles formed from colinear points, or overlapping triangles.

Attributes

- `edges`
- `neighbors`
- `is_delaunay`(bool) Whether the Triangulation is a calculated Delaunay triangulation (where triangles was not specified) or not.

`calculate_plane_coefficients(z)`

Calculate plane equation coefficients for all unmasked triangles from the point (x,y) coordinates and specified z-array of shape (npoints). Returned array has shape (npoints,3) and allows z-value at (x,y) position in triangle tri to be calculated using $z = \text{array}[\text{tri,0}]^*x + \text{array}[\text{tri,1}]^*y + \text{array}[\text{tri,2}]$. 

1599
edges
Return integer array of shape (nedges,2) containing all edges of non-masked triangles.

Each edge is the start point index and end point index. Each edge (start,end and end,start)
appears only once.

static get_from_args_and_kwargs(*args, **kwargs)
Return a Triangulation object from the args and kwargs, and the remaining args and kwags
with the consumed values removed.

There are two alternatives: either the first argument is a Triangulation object, in which case it
is returned, or the args and kwargs are sufficient to create a new Triangulation to return. In the
latter case, see Triangulation.__init__ for the possible args and kwargs.

get_masked_triangles()
Return an array of triangles that are not masked.

get_trifinder()
Return the default matplotlib.tri.TriFinder of this triangulation, creating it if necessary.
This allows the same TriFinder object to be easily shared.

neighbors
Return integer array of shape (ntri,3) containing neighbor triangles.

For each triangle, the indices of the three triangles that share the same edges, or -1 if there is
no such neighboring triangle. neighbors[i,j] is the triangle that is the neighbor to the edge from
point index triangles[i,j] to point index triangles[i,(j+1)%3].

set_mask(mask)
Set or clear the mask array. This is either None, or a boolean array of shape (ntri).

class matplotlib.tri.TriFinder(triangulation)
Abstract base class for classes used to find the triangles of a Triangulation in which (x,y) points lie.

Rather than instantiate an object of a class derived from TriFinder, it is usually better to use the
function matplotlib.tri.Triangulation.get_trifinder().

Derived classes implement __call__(x,y) where x,y are array_like point coordinates of the same shape.

class matplotlib.tri.TrapezoidMapTriFinder(triangulation)
Bases: matplotlib.tri.trifinder.TriFinder

TriFinder class implemented using the trapezoid map algorithm from the book “Computational Ge-
ometry, Algorithms and Applications”, second edition, by M. de Berg, M. van Kreveld, M. Overmars
and O. Schwarzkopf.

The triangulation must be valid, i.e. it must not have duplicate points, triangles formed from colinear
points, or overlapping triangles. The algorithm has some tolerance to triangles formed from colinear
points, but this should not be relied upon.

__call__(x, y)
Return an array containing the indices of the triangles in which the specified x,y points lie, or
-1 for points that do not lie within a triangle.

x, y are array_like x and y coordinates of the same shape and any number of dimensions.
Returns integer array with the same shape and \( x \) and \( y \).

```python
class matplotlib.tri.TriInterpolator(triangulation, z, trifinder=None)
```
Abstract base class for classes used to perform interpolation on triangular grids.

Derived classes implement the following methods:
- `__call__(x, y)` , where \( x, y \) are array_like point coordinates of the same shape, and that returns a masked array of the same shape containing the interpolated \( z \)-values.
- `gradient(x, y)` , where \( x, y \) are array_like point coordinates of the same shape, and that returns a list of 2 masked arrays of the same shape containing the 2 derivatives of the interpolator (derivatives of interpolated \( z \) values with respect to \( x \) and \( y \)).

```python
class matplotlib.tri.LinearTriInterpolator(triangulation, z, trifinder=None)
```
Bases: `matplotlib.tri.triinterpolate.TriInterpolator`

A LinearTriInterpolator performs linear interpolation on a triangular grid.

Each triangle is represented by a plane so that an interpolated value at point \((x,y)\) lies on the plane of the triangle containing \((x,y)\). Interpolated values are therefore continuous across the triangulation, but their first derivatives are discontinuous at edges between triangles.

**Parameters**
- `triangulation` : `Triangulation` object
  The triangulation to interpolate over.
- `z` : array_like of shape \((\text{npoints},)\)
  Array of values, defined at grid points, to interpolate between.
- `trifinder` : `TriFinder` object, optional
  If this is not specified, the Triangulation’s default TriFinder will be used by calling `matplotlib.tri.Triangulation.get_trifinder()`.

**Methods**

```python
__call__(x, y)  # Returns interpolated values at x,y points
gradient(x, y) # Returns interpolated derivatives at x,y points
```

**__call__(x, y)**

Returns a masked array containing interpolated values at the specified \( x, y \) points.

**Parameters**
- \( x, y \) : array-like
  \( x \) and \( y \) coordinates of the same shape and any number of dimensions.

**Returns**
- \( z \) : `np.ma.array`
  Masked array of the same shape as \( x \) and \( y \); values corresponding to \((x, y)\) points outside of the triangulation are masked out.

**gradient(x, y)**

Returns a list of 2 masked arrays containing interpolated derivatives at the specified \( x, y \) points.

**Parameters**
- \( x, y \) : array-like
  \( x \) and \( y \) coordinates of the same shape and any number of dimensions.

**Returns**
- \( dzdx, dzdy \) : `np.ma.array`
2 masked arrays of the same shape as $x$ and $y$; values corresponding to $(x,y)$ points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

class matplotlib.tri.CubicTriInterpolator(triangulation, z, kind=\'u\'min\'_E\', trifinder=None, dz=None)

Bases: matplotlib.tri.triinterpolate.TriInterpolator

A CubicTriInterpolator performs cubic interpolation on triangular grids.

In one-dimension - on a segment - a cubic interpolating function is defined by the values of the function and its derivative at both ends. This is almost the same in 2-d inside a triangle, except that the values of the function and its 2 derivatives have to be defined at each triangle node.

The CubicTriInterpolator takes the value of the function at each node - provided by the user - and internally computes the value of the derivatives, resulting in a smooth interpolation. (As a special feature, the user can also impose the value of the derivatives at each node, but this is not supposed to be the common usage.)

**Parameters**

- **triangulation**: Triangulation object
  The triangulation to interpolate over.
- **z**: array_like of shape (npoints,)
  Array of values, defined at grid points, to interpolate between.
- **kind**: \{'min\'_E\', \'geom\', \'user\'\}, optional
  Choice of the smoothing algorithm, in order to compute the interpolant derivatives (defaults to \'min\'_E\'):
  - if \'min\'_E\': (default) The derivatives at each node is computed to minimize a bending energy.
  - if \'geom\': The derivatives at each node is computed as a weighted average of relevant triangle normals. To be used for speed optimization (large grids).
  - if \'user\': The user provides the argument dz, no computation is hence needed.
- **trifinder**: TriFinder object, optional
  If not specified, the Triangulation’s default TriFinder will be used by calling matplotlib.tri.Triangulation.get_trifinder().
- **dz**: tuple of array_likes (dzdx, dzdy), optional
  Used only if **kind** = \'user\'. In this case dz must be provided as (dzdx, dzdy) where dzdx, dzdy are arrays of the same shape as z and are the interpolant first derivatives at the triangulation points.

**Notes**

This note is a bit technical and details the way a CubicTriInterpolator computes a cubic interpolation.

The interpolation is based on a Clough-Tocher subdivision scheme of the triangulation mesh (to make it clearer, each triangle of the grid will be divided in 3 child-triangles, and on each child triangle the interpolated function is a cubic polynomial of the 2 coordinates). This technique originates from
FEM (Finite Element Method) analysis; the element used is a reduced Hsieh-Clough-Tocher (HCT) element. Its shape functions are described in [R1]. The assembled function is guaranteed to be C1-smooth, i.e. it is continuous and its first derivatives are also continuous (this is easy to show inside the triangles but is also true when crossing the edges).

In the default case (kind = 'min_E'), the interpolant minimizes a curvature energy on the functional space generated by the HCT element shape functions - with imposed values but arbitrary derivatives at each node. The minimized functional is the integral of the so-called total curvature (implementation based on an algorithm from [R2] - PCG sparse solver):

\[
E(z) = \frac{1}{2} \int_{\Omega} \left( \left( \frac{\partial^2 z}{\partial x^2} \right)^2 + \left( \frac{\partial^2 z}{\partial y^2} \right)^2 + 2 \left( \frac{\partial^2 z}{\partial y \partial x} \right)^2 \right) \, dx \, dy
\]  

(70.1)

If the case kind = 'geom’ is chosen by the user, a simple geometric approximation is used (weighted average of the triangle normal vectors), which could improve speed on very large grids.

References

[R1], [R2]

Methods

__call__(x, y)  (Returns interpolated values at x,y points)
gradient(x, y)  (Returns interpolated derivatives at x,y points)

d__call__ (x, y)

Returns a masked array containing interpolated values at the specified x,y points.

Parameters x, y : array-like
x and y coordinates of the same shape and any number of dimensions.

Returns z : np.ma.array
Masked array of the same shape as x and y ; values corresponding to (x, y) points outside of the triangulation are masked out.

dgradient (x, y)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified x,y points.

Parameters x, y : array-like
x and y coordinates of the same shape and any number of dimensions.

Returns dzdx, dzdy : np.ma.array
2 masked arrays of the same shape as x and y ; values corresponding to (x,y) points outside of the triangulation are masked out. The first returned array contains the values of \( \frac{\partial z}{\partial x} \) and the second those of \( \frac{\partial z}{\partial y} \).
Examples

An example of effective application is shown below (plot of the direction of the vector field derivated from a known potential field):

class matplotlib.tri.TriRefiner(triangulation)

Abstract base class for classes implementing mesh refinement.

A TriRefiner encapsulates a Triangulation object and provides tools for mesh refinement and interpolation.

Derived classes must implements:

- **refine_triangulation**(return_tri_index=False, **kwargs)**, where the optional keyword arguments **kwargs** are defined in each TriRefiner concrete implementation, and which returns:
  - a refined triangulation
  - optionally (depending on return_tri_index), for each point of the refined triangulation: the index of the initial triangulation triangle to which it belongs.

- **refine_field**(z, triinterpolator=None, **kwargs)**, where:
  - z array of field values (to refine) defined at the base triangulation nodes
  - triinterpolator is a TriInterpolator (optional)
  - the other optional keyword arguments **kwargs** are defined in each TriRefiner concrete implementation

and which returns (as a tuple) a refined triangular mesh and the interpolated values of the field

1604 Chapter 70. triangular grids
at the refined triangulation nodes.

class matplotlib.tri.UniformTriRefiner(triangulation)
   Bases: matplotlib.tri.trirefine.TriRefiner

Uniform mesh refinement by recursive subdivisions.

   Parameters triangulation : Triangulation
      The encapsulated triangulation (to be refined)

   refine_field(z, triinterpolator=None, subdiv=3)
      Refines a field defined on the encapsulated triangulation.

      Returns refi_tri (refined triangulation), refi_z (interpolated values of the field at the node of the refined triangulation).

      Parameters z : 1d-array-like of length n_points
         Values of the field to refine, defined at the nodes of the encapsulated triangulation. (n_points is the number of points in the initial triangulation)

      triinterpolator : TriInterpolator, optional
         Interpolator used for field interpolation. If not specified, a CubicTriInterpolator will be used.

      subdiv : integer, optional
         Recursion level for the subdivision. Defaults to 3. Each triangle will be divided into $4^{**\text{subdiv}}$ child triangles.

      Returns refi_tri : Triangulation object
         The returned refined triangulation

      refi_z : 1d array of length: refi_tri node count.
         The returned interpolated field (at refi_tri nodes)

Examples

The main application of this method is to plot high-quality iso-contours on a coarse triangular grid (e.g., triangulation built from relatively sparse test data):
**refine_triangulation**(*return_tri_index=False, subdiv=3*)

Computes an uniformly refined triangulation `refi_triangulation` of the encapsulated triangulation.

This function refines the encapsulated triangulation by splitting each father triangle into 4 child sub-triangles built on the edges midside nodes, recursively (level of recursion `subdiv`). In the end, each triangle is hence divided into $4^{subdiv}$ child triangles. The default value for `subdiv` is 3 resulting in 64 refined subtriangles for each triangle of the initial triangulation.

**Parameters**

- `return_tri_index` : boolean, optional
  Boolean indicating whether an index table indicating the father triangle index of each point will be returned. Default value False.

- `subdiv` : integer, optional
  Recursion level for the subdivision. Defaults value 3. Each triangle will be divided into $4^{subdiv}$ child triangles.

**Returns**

- `refi_triangulation` : `Triangulation`
  The returned refined triangulation

- `found_index` : array-like of integers
  Index of the initial triangulation containing triangle, for each point of `refi_triangulation`. Returned only if `return_tri_index` is set to True.

**class** `matplotlib.tri.TriAnalyzer`(triangulation)

Define basic tools for triangular mesh analysis and improvement.
A TriAnalysis encapsulates a Triangulation object and provides basic tools for mesh analysis and mesh improvement.

**Parameters**

**triangulation**: Triangulation object

The encapsulated triangulation to analyze.

**Attributes**

- **scale_factors**

- **circle_ratios**(rescale=True)

  Returns a measure of the triangulation triangles flatness.
  
  The ratio of the incircle radius over the circumcircle radius is a widely used indicator of a triangle flatness. It is always <= 0.5 and == 0.5 only for equilateral triangles. Circle ratios below 0.01 denote very flat triangles.

  To avoid unduly low values due to a difference of scale between the 2 axis, the triangular mesh can first be rescaled to fit inside a unit square with scale_factors (Only if rescale is True, which is its default value).

  **Parameters**

  **rescale**: boolean, optional
  
  If True, a rescaling will be internally performed (based on scale_factors, so that the (unmasked) triangles fit exactly inside a unit square mesh. Default is True.

  **Returns**

  **circle_ratios**: masked array
  
  Ratio of the incircle radius over the circumcircle radius, for each 'rescaled' triangle of the encapsulated triangulation. Values corresponding to masked triangles are masked out.

- **get_flat_tri_mask**(min_circle_ratio=0.01, rescale=True)

  Eliminates excessively flat border triangles from the triangulation.

  Returns a mask new_mask which allows to clean the encapsulated triangulation from its border-located flat triangles (according to their circle_ratios()). This mask is meant to be subsequently applied to the triangulation using matplotlib.tri.Triangulation.set_mask() . new_mask is an extension of the initial triangulation mask in the sense that an initially masked triangle will remain masked.

  The new_mask array is computed recursively; at each step flat triangles are removed only if they share a side with the current mesh border. Thus no new holes in the triangulated domain will be created.

  **Parameters**

  **min_circle_ratio**: float, optional
  
  Border triangles with incircle/circumcircle radii ratio r/R will be removed if r/R < min_circle_ratio. Default value: 0.01

  **rescale**: boolean, optional
  
  If True, a rescaling will first be internally performed (based on scale_factors), so that the (unmasked) triangles fit exactly inside a unit square mesh. This rescaling accounts for the difference of scale which might exist between the 2 axis. Default (and recommended) value is True.

  **Returns**

  **new_mask**: array-like of booleans
Mask to apply to encapsulated triangulation. All the initially masked triangles remain masked in the `new_mask`.

**Notes**

The rationale behind this function is that a Delaunay triangulation - of an unstructured set of points - sometimes contains almost flat triangles at its border, leading to artifacts in plots (especially for high-resolution contouring). Masked with computed `new_mask`, the encapsulated triangulation would contain no more unmasked border triangles with a circle ratio below `min_circle_ratio`, thus improving the mesh quality for subsequent plots or interpolation.

**Examples**

Please refer to the following illustrating example:

```
Filtering a Delaunay mesh
(application to high-resolution tricontouring)
```

**scale_factors**

Factors to rescale the triangulation into a unit square.

Returns $k$, tuple of 2 scale factors.

**Returns** $k$ : tuple of 2 floats (kx, ky)

Tuple of floats that would rescale the triangulation:

$[\text{triangulation.x} \ast \text{kx}, \text{triangulation.y} \ast \text{ky}]$
fits exactly inside a unit square.
71.1 matplotlib.type1font

This module contains a class representing a Type 1 font.

This version reads pfa and pfb files and splits them for embedding in pdf files. It also supports SlantFont and ExtendFont transformations, similarly to pdfTeX and friends. There is no support yet for subsetting.

Usage:

```python
>>> font = Type1Font(filename)
>>> clear_part, encrypted_part, finale = font.parts
>>> slanted_font = font.transform({'slant': 0.167})
>>> extended_font = font.transform({'extend': 1.2})
```

Sources:

- Adobe Technical Note #5040, Supporting Downloadable PostScript Language Fonts.

class matplotlib.type1font.Type1Font(input)

Bases: object

A class representing a Type-1 font, for use by backends.

 partes

A 3-tuple of the cleartext part, the encrypted part, and the finale of zeros.

prop

A dictionary of font properties.

Initialize a Type-1 font. input can be either the file name of a pfb file or a 3-tuple of already-decoded Type-1 font parts.

 partes

prop
transform(effects)

Transform the font by slanting or extending. effects should be a dict where effects['slant'] is the tangent of the angle that the font is to be slanted to the right (so negative values slant to the left) and effects['extend'] is the multiplier by which the font is to be extended (so values less than 1.0 condense). Returns a new Type1Font object.
72.1 matplotlib.units

The classes here provide support for using custom classes with matplotlib, e.g., those that do not expose the array interface but know how to converter themselves to arrays. It also supports classes with units and units conversion. Use cases include converters for custom objects, e.g., a list of datetime objects, as well as for objects that are unit aware. We don’t assume any particular units implementation, rather a units implementation must provide a ConversionInterface, and the register with the Registry converter dictionary. For example, here is a complete implementation which supports plotting with native datetime objects:

```python
import matplotlib.units as units
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import datetime

class DateConverter(units.ConversionInterface):
    @staticmethod
    def convert(value, unit, axis):
        'convert value to a scalar or array'
        return dates.date2num(value)

    @staticmethod
    def axisinfo(unit, axis):
        'return the major and minor tick locators and formatters'
        if unit != 'date': return None
        majloc = dates.AutoDateLocator()
        majfmt = dates.AutoDateFormatter(majloc)
        return AxisInfo(majloc=majloc,
                        majfmt=majfmt,
                        label='date')

    @staticmethod
    def default_units(x, axis):
        'return the default unit for x or None'
        return 'date'

# finally we register our object type with a converter
units.registry[datetime.date] = DateConverter()
```
class matplotlib.units.AxisInfo(majloc=None, minloc=None, majfmt=None, minfmt=None, label=None, default_limits=None)

information to support default axis labeling and tick labeling, and default limits

majloc and minloc: TickLocators for the major and minor ticks majfmt and minfmt: TickFormatters for the major and minor ticks label: the default axis label default_limits: the default min, max of the axis if no data is present If any of the above are None, the axis will simply use the default

class matplotlib.units.ConversionInterface

The minimal interface for a converter to take custom instances (or sequences) and convert them to values mpl can use

static axisinfo(unit, axis)

return an units.AxisInfo instance for axis with the specified units

static convert(obj, unit, axis)

convert obj using unit for the specified axis. If obj is a sequence, return the converted sequence. The output must be a sequence of scalars that can be used by the numpy array layer

static default_units(x, axis)

return the default unit for x or None for the given axis

static is_numlike(x)

The matplotlib datalim, autoscaling, locators etc work with scalars which are the units converted to floats given the current unit. The converter may be passed these floats, or arrays of them, even when units are set. Derived conversion interfaces may opt to pass plain-ol unitless numbers through the conversion interface and this is a helper function for them.

class matplotlib.units.Registry

Bases: dict

register types with conversion interface

get_converter(x)

get the converter interface instance for x, or None
73.1 matplotlib.widgets

73.1.1 GUI Neutral widgets

Widgets that are designed to work for any of the GUI backends. All of these widgets require you to pre-
define an matplotlib.axes.Axes instance and pass that as the first arg. matplotlib doesn’t try to be too smart with respect to layout – you will have to figure out how wide and tall you want your Axes to be to accommodate your widget.

```python
class matplotlib.widgets.AxesWidget(ax)
    Bases: matplotlib.widgets.Widget

    Widget that is connected to a single Axes.

    To guarantee that the widget remains responsive and not garbage-collected, a reference to the object should be maintained by the user.

    This is necessary because the callback registry maintains only weak-refs to the functions, which are member functions of the widget. If there are no references to the widget object it may be garbage collected which will disconnect the callbacks.

    Attributes:
    ax [Axes] The parent axes for the widget  
    canvas [FigureCanvasBase subclass] The parent figure canvas for the widget.  
    active [bool] If False, the widget does not respond to events.  
    connect_event(event, callback)
        Connect callback with an event.

        This should be used in lieu of figure.canvas.mpl_connect since this function stores callback ids for later clean up.

    disconnect_events()
        Disconnect all events created by this widget.

    ignore(event)
        Return True if event should be ignored.

        This method (or a version of it) should be called at the beginning of any event callback.
```
class matplotlib.widgets.Button(ax, label='', image=None, color='0.85', hovercolor='0.95')
Bases: matplotlib.widgets.AxesWidget

A GUI neutral button.

For the button to remain responsive you must keep a reference to it.

The following attributes are accessible

- **ax** The matplotlib.axes.Axes instance the button renders into.
- **label** A matplotlib.text.Text instance.
- **color** The color of the button when not hovering.
- **hovercolor** The color of the button when hovering.

Call on_clicked() to connect to the button

**Parameters**

- **ax**: matplotlib.axes.Axes
  The matplotlib.axes.Axes instance the button will be placed into.
- **label**: str
  The button text. Accepts string.
- **image**: array, mpl image, PIL image
  The image to place in the button, if not None. Can be any legal arg to imshow (numpy array, matplotlib Image instance, or PIL image).
- **color**: color
  The color of the button when not activated
- **hovercolor**: color
  The color of the button when the mouse is over it

**disconnect**(cid)
remove the observer with connection id cid

**on_clicked**(func)
When the button is clicked, call this func with event
A connection id is returned which can be used to disconnect

class matplotlib.widgets.CheckButtons(ax, labels, actives)
Bases: matplotlib.widgets.AxesWidget

A GUI neutral radio button.

For the check buttons to remain responsive you much keep a reference to this object.

The following attributes are exposed

- **ax** The matplotlib.axes.Axes instance the buttons are located in
- **labels** List of matplotlib.text.Text instances
- **lines** List of (line1, line2) tuples for the x’s in the check boxes. These lines exist for each box, but have set_visible(False) when its box is not checked.
- **rectangles** List of matplotlib.patches.Rectangle instances

Connect to the CheckButtons with the on_clicked() method

Add check buttons to matplotlib.axes.Axes instance ax

**Parameters**

- **labels**: A len(buttons) list of labels as strings
- **actives**: A len(buttons) list of booleans indicating whether the button is active

**disconnect**(cid)
remove the observer with connection id cid
**on_clicked**(*func*)
When the button is clicked, call *func* with button label

A connection id is returned which can be used to disconnect

```python
class matplotlib.widgets.Cursor(ax, horizOn=True, vertOn=True, useblit=False, **lineprops)
```

Bases: `matplotlib.widgets.AxesWidget`

A horizontal and vertical line span the axes that and move with the pointer. You can turn off the hline or vline spectively with the attributes

- `horizOn` Controls the visibility of the horizontal line
- `vertOn` Controls the visibility of the horizontal line

and the visibility of the cursor itself with the `visible` attribute.

For the cursor to remain responsive you much keep a reference to it.

Add a cursor to `ax`. If `useblit=True`, use the backend-dependent blitting features for faster updates (GTKAgg only for now). `lineprops` is a dictionary of line properties.

```python
clear(event)
clear the cursor
```

```python
onmove(event)
on move motion draw the cursor if visible
```

```python
class matplotlib.widgets.Lasso(ax, xy, callback=None, useblit=True)
```

Bases: `matplotlib.widgets.AxesWidget`

73.1. `matplotlib.widgets`
Selection curve of an arbitrary shape.

The selected path can be used in conjunction with `contains_point()` to select data points from an image.

Unlike `LassoSelector`, this must be initialized with a starting point `xy`, and the `Lasso` events are destroyed upon release.

Parameters:
- `xy` [array] Coordinates of the start of the lasso.
- `callback` [function] Whenever the lasso is released, the callback function is called and passed the vertices of the selected path.

Example usage:

```python
ax = subplot(111)
ax.plot(x, y)

def onselect(verts):
    print(verts)
lasso = LassoSelector(ax, onselect)
```

1618 Chapter 73. widgets
onrelease(event)

update_background(event)

class matplotlib.widgets.LockDraw
Some widgets, like the cursor, draw onto the canvas, and this is not desirable under all circumstances, like when the toolbar is in zoom-to-rect mode and drawing a rectangle. The module level “lock” allows someone to grab the lock and prevent other widgets from drawing. Use matplotlib.widgets.lock(someobj) to pr

available(o)
  drawing is available to o

isowner(o)
  Return True if o owns this lock

locked()
  Return True if the lock is currently held by an owner

release(o)
  release the lock

class matplotlib.widgets.MultiCursor(canvas, axes, useblit=True, horizOn=False, vertOn=True, **lineprops)
Bases: matplotlib.widgets.Widget
Provide a vertical (default) and/or horizontal line cursor shared between multiple axes.
For the cursor to remain responsive you much keep a reference to it.
Example usage:

from matplotlib.widgets import MultiCursor
from pylab import figure, show, np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)
fig = figure()
ax1 = fig.add_subplot(211)
ax1.plot(t, s1)

ax2 = fig.add_subplot(212, sharex=ax1)
ax2.plot(t, s2)

multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1, horizOn=False, vertOn=True)
show()

clear(event)
  clear the cursor
connect()
   connect events

disconnect()
   disconnect events

onmove(event)

class matplotlib.widgets.RadioButtons(ax, labels, active=0, activecolor='u'blue')
   Bases: matplotlib.widgets.AxesWidget

A GUI neutral radio button

For the buttons to remain responsive you much keep a reference to this object.

The following attributes are exposed
   ax The matplotlib.axes.Axes instance the buttons are in
   activecolor The color of the button when clicked
   labels A list of matplotlib.text.Text instances
   circles A list of matplotlib.patches.Circle instances

Connect to the RadioButtons with the on_clicked() method

Add radio buttons to matplotlib.axes.Axes instance ax
labels A len(buttons) list of labels as strings
active The index into labels for the button that is active
activecolor The color of the button when clicked
disconnect(cid)
   remove the observer with connection id cid

on_clicked(func)
   When the button is clicked, call func with button label

   A connection id is returned which can be used to disconnect

class matplotlib.widgets.RectangleSelector(ax, onselect, drawtype='u'box',
   minspanx=None, minspany=None, useblit=False, lineprops=None, rectprops=None,
   spancoords='u'data', button=None)

   Bases: matplotlib.widgets.AxesWidget

Select a rectangular region of an axes.

For the cursor to remain responsive you much keep a reference to it.

Example usage:

   from matplotlib.widgets import RectangleSelector
   from pylab import *

   def onselect(eclick, erelease):
      'eclick and erelease are matplotlib events at press and release'
      print 'startposition : (%f, %f)' % (eclick.xdata, eclick.ydata)
      print 'endposition : (%f, %f)' % (erelease.xdata, erelease.ydata)
      print 'used button : ', eclick.button
```python
def toggle_selector(event):
    print 'Key pressed.'
    if event.key in ['Q', 'q'] and toggle_selector.RS.active:
        print 'RectangleSelector deactivated.'
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.RS.active:
        print 'RectangleSelector activated.'
        toggle_selector.RS.set_active(True)

x = arange(100)/(99.0)
y = sin(x)
fig = figure
ax = subplot(111)
ax.plot(x, y)

toggle_selector.RS = RectangleSelector(ax, onselect, drawtype='line')
connect('key_press_event', toggle_selector)
show()
```

Create a selector in `ax`. When a selection is made, clear the span and call `onselect` with:

```python
onselect(pos_1, pos_2)
```

and clear the drawn box/line. The `pos_1` and `pos_2` are arrays of length 2 containing the x- and y-coordinate.

If `minspanx` is not `None` then events smaller than `minspanx` in x direction are ignored (it’s the same for y).

The rectangle is drawn with `rectprops`; default:

```python
rectprops = dict(facecolor='red', edgcolor = 'black',
                 alpha=0.5, fill=False)
```

The line is drawn with `lineprops`; default:

```python
lineprops = dict(color='black', linestyle='-',
                  linewidth = 2, alpha=0.5)
```

Use `drawtype` if you want the mouse to draw a line, a box or nothing between click and actual position by setting

```python
drawtype = 'line', drawtype='box' or drawtype = 'none'.
```

`spancoords` is one of ‘data’ or ‘pixels’. If ‘data’, `minspanx` and `minspanx` will be interpreted in the same coordinates as the x and y axis. If ‘pixels’, they are in pixels.

`button` is a list of integers indicating which mouse buttons should be used for rectangle selection. You can also specify a single integer if only a single button is desired. Default is `None`, which does not limit which button can be used.

**Note, typically:** 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button
get_active()
Get status of active mode (boolean variable)

ignore(event)
return True if event should be ignored

onmove(event)
on motion notify event if box/line is wanted

press(event)
on button press event

release(event)
on button release event

set_active(active)
Use this to activate / deactivate the RectangleSelector from your program with an boolean parameter active.

update()
draw using newfangled blit or oldfangled draw depending on useblit

update_background(event)
force an update of the background

class matplotlib.widgets.Slider(ax, label, valmin, valmax, valinit=0.5, valfmt=u’%1.2f’,
closedmin=True, closedmax=True, slidermin=0, slidermax=None, dragging=True, **kwargs)
Bases: matplotlib.widgets.AxesWidget

A slider representing a floating point range.

For the slider to remain responsive you must maintain a reference to it.

The following attributes are defined

val : the current slider value
vline [a matplotlib.lines.Line2D instance] representing the initial value of the slider
poly [A matplotlib.patches.Polygon instance] which is the slider knob
valfmt : the format string for formatting the slider text
label [a matplotlib.text.Text instance] for the slider label
closedmin : whether the slider is closed on the minimum

closedmax : whether the slider is closed on the maximum
slidermin [another slider - if not None, this slider must be] greater than slidermin
slidermax [another slider - if not None, this slider must be] less than slidermax

dragging : allow for mouse dragging on slider

Call on_changed() to connect to the slider event

Create a slider from valmin to valmax in axes ax.

additional kwargs are passed on to self.poly which is the matplotlib.patches.Rectangle
which draws the slider knob. See the matplotlib.patches.Rectangle documentation valid prop-

erty names (e.g., facecolor, edgecolor, alpha, ...)

Parameters

ax : Axes
The Axes to put the slider in
label : str
    Slider label
valmin : float
    The minimum value of the slider
valmax : float
    The maximum value of the slider
valinit : float
    The slider initial position
label : str
    The slider label
valfmt : str
    Used to format the slider value, fprint format string
closedmin : bool
    Indicate whether the slider interval is closed on the bottom
closedmax : bool
    Indicate whether the slider interval is closed on the top
slidermin : Slider or None
    Do not allow the current slider to have a value less than slidermin
slidermax : Slider or None
    Do not allow the current slider to have a value greater than slidermax
dragging : bool
    if the slider can be dragged by the mouse
disconnect(cid)
    remove the observer with connection id cid
on_changed(func)
    When the slider value is changed, call func with the new slider position
    A connection id is returned which can be used to disconnect
reset()
    reset the slider to the initial value if needed
set_val(val)

class matplotlib.widgets.SpanSelector(ax, onselect, direction, minspan=None, useblit=False, rectprops=None, onmove_callback=None, span_stays=False)
Bases: matplotlib.widgets.AxesWidget
Select a min/max range of the x or y axes for a matplotlib Axes.
For the selector to remain responsive you much keep a reference to it.
Example usage:

    ax = subplot(111)
    ax.plot(x, y)
    
    def onselect(vmin, vmax):

73.1. matplotlib.widgets
print vmin, vmax
span = SpanSelector(ax, onselect, 'horizontal')

_onmove_callback_is_an_optional_callback_that_is_called_on_move_within_the_span_range
Create a span selector in `ax`. When a selection is made, clear the span and call `onselect` with:

```python
onselect(vmin, vmax)
```

and clear the span.

direction must be 'horizontal' or 'vertical'

If `minspan` is not None, ignore events smaller than `minspan`
The span rectangle is drawn with `rectprops`; default:
```
rectprops = dict(facecolor='red', alpha=0.5)
```
Set the visible attribute to `False` if you want to turn off the functionality of the span selector.

If `span_stays` is True, the span stays visible after making a valid selection.

```python
ignore(event)
    return True if event should be ignored
```

```python
new_axes(ax)
```

```python
onmove(event)
    on motion notify event
```

```python
press(event)
    on button press event
```

```python
release(event)
    on button release event
```

```python
update()
    Draw using newfangled blit or oldfangled draw depending on `useblit`
```

```python
update_background(event)
    force an update of the background
```

class matplotlib.widgets.SubplotTool(targetfig, toolfig)
Bases: matplotlib.widgets.Widget

A tool to adjust to subplot params of a `matplotlib.figure.Figure`

`targetfig` The figure instance to adjust
`toolfig` The figure instance to embed the subplot tool into. If None, a default figure will be created. If you are using this from the GUI

```python
funcbottom(val)
```

```python
funcspace(val)
```
funcleft(val)

funcright(val)

functop(val)

funcwspace(val)

class matplotlib.widgets.Widget
    Bases: object
    
    Abstract base class for GUI neutral widgets

drawn = True

eventson = True
Part IX

Matplotlib Examples
74.1 animation example code: animate_decay.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def data_gen():
    t = data_gen.t
    cnt = 0
    while cnt < 1000:
        cnt+=1
        t += 0.05
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)
    data_gen.t = 0

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.set_ylim(-1.1, 1.1)
ax.set_xlim(0, 5)
ax.grid()
xdata, ydata = [], []

def run(data):
    # update the data
    t,y = data
    xdata.append(t)
ydata.append(y)
xmin, xmax = ax.get_xlim()
    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)
    return line,

ani = animation.FuncAnimation(fig, run, data_gen, blit=True, interval=10,
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def update_line(num, data, line):
    line.set_data(data[...,:num])
    return line,

fig1 = plt.figure()
data = np.random.rand(2, 25)
l, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l),
    interval=50, blit=True)
#line_ani.save('lines.mp4')

fig2 = plt.figure()
x = np.arange(-9, 10)
y = np.arange(-9, 10).reshape(-1, 1)
base = np.hypot(x, y)
ims = []
for add in np.arange(15):
    ims.append((plt.pcolor(x, y, base + add, norm=plt.Normalize(0, 30)),))

im_ani = animation.ArtistAnimation(fig2, ims, interval=50, repeat_delay=3000,
    blit=True)
#im_ani.save('im.mp4', metadata={'artist':'Guido'})

plt.show()
# Same as basic_example, but writes files using a single MovieWriter instance
# without putting on screen
# -*- noplot -*-
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def update_line(num, data, line):
    line.set_data(data[...,:num])
    return

    
# Set up formatting for the movie files
Writer = animation.writers['ffmpeg']
writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)

fig1 = plt.figure()
data = np.random.rand(2, 25)
l, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l),
interval=50, blit=True)
line_ani.save('lines.mp4', writer=writer)

fig2 = plt.figure()
x = np.arange(-9, 10)
y = np.arange(-9, 10).reshape(-1, 1)
base = np.hypot(x, y)
ims = []
for add in np.arange(15):
    ims.append((plt.pcolor(x, y, base + add, norm=plt.Normalize(0, 30)),))

im_ani = animation.ArtistAnimation(fig2, ims, interval=50, repeat_delay=3000,
blit=True)
im_ani.save('im.mp4', writer=writer)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.4 animation example code: bayes_update.py

[source code]
# update a distribution based on new data.
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as ss
from matplotlib.animation import FuncAnimation

class UpdateDist(object):
    def __init__(self, ax, prob=0.5):
        self.success = 0
        self.prob = prob
        self.line, = ax.plot([], [], 'k-')
        self.x = np.linspace(0, 1, 200)
        self.ax = ax

        # Set up plot parameters
        self.ax.set_xlim(0, 1)
        self.ax.set_ylim(0, 15)
        self.ax.grid(True)

        # This vertical line represents the theoretical value, to
        # which the plotted distribution should converge.
        self.ax.axvline(prob, linestyle='--', color='black')

    def init(self):
        self.success = 0
        self.line.set_data([], [])
        return self.line,

    def __call__(self, i):
        # This way the plot can continuously run and we just keep
        # watching new realizations of the process
        if i == 0:
            return self.init()

        # Choose success based on exceed a threshold with a uniform pick
        if np.random.rand(1,) < self.prob:
            self.success += 1
        y = ss.beta.pdf(self.x, self.success + 1, (i - self.success) + 1)
        self.line.set_data(self.x, y)
        return self.line,

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ud = UpdateDist(ax, prob=0.7)
anim = FuncAnimation(fig, ud, frames=np.arange(100), init_func=ud.init,
                     interval=100, blit=True)
plt.show()
# Double pendulum formula translated from the C code at

from numpy import sin, cos, pi, array
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.0  # length of pendulum 1 in m
L2 = 1.0  # length of pendulum 2 in m
M1 = 1.0  # mass of pendulum 1 in kg
M2 = 1.0  # mass of pendulum 2 in kg

def derivs(state, t):
    dydx = np.zeros_like(state)
    dydx[0] = state[1]
    del_ = state[2] - state[0]
    den1 = (M1 + M2)*L1 - M2*L1*cos(del_)*cos(del_)
    dydx[1] = (M2*L1*state[1]*state[1]*sin(del_)*cos(del_)
               + M2*G*sin(state[2])*cos(del_)
               + M2*L2*state[3]*state[3]*sin(del_)
               - (M1 + M2)*G*sin(state[0]))/den1

    den2 = (L2/L1)*den1
               + (M1 + M2)*G*sin(state[0])*cos(del_)
               - (M1 + M2)*L1*state[1]*state[1]*sin(del_)
               - (M1 + M2)*G*sin(state[2]))/den2

    return dydx

# create a time array from 0..100 sampled at 0.05 second steps
dt = 0.05
T = np.arange(0.0, 20, dt)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)
th1 = 120.0
w1 = 0.0
th2 = -10.0
w2 = 0.0
rad = pi/180
# initial state
state = np.array([th1, w1, th2, w2])*pi/180.

# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)

x1 = L1*sin(y[:,0])
y1 = -L1*cos(y[:,0])

x2 = L2*sin(y[:,2]) + x1
y2 = -L2*cos(y[:,2]) + y1

fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)

def init():
    line.set_data([], [])
time_text.set_text('')
    return line, time_text

def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    line.set_data(thisx, thisy)
time_text.set_text(time_template%(i*dt))
    return line, time_text

ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
                               interval=25, blit=True, init_func=init)

#ani.save('double_pendulum.mp4', fps=15)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.6 animation example code: dynamic_image.py

[source code]

#!/usr/bin/env python

"""
An animated image
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()

def f(x, y):
    return np.sin(x) + np.cos(y)

x = np.linspace(0, 2 * np.pi, 120)
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)

im = plt.imshow(f(x, y), cmap=plt.get_cmap('jet'))

def updatefig(*args):
    global x, y
    x += np.pi / 15.
y += np.pi / 20.
im.set_array(f(x, y))
    return im,

ani = animation.FuncAnimation(fig, updatefig, interval=50, blit=True)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.7 animation example code: dynamic_image2.py

[source code]

#!/usr/bin/env python

An animated image

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()

def f(x, y):
    return np.sin(x) + np.cos(y)

x = np.linspace(0, 2 * np.pi, 120)
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)

# ims is a list of lists, each row is a list of artists to draw in the current frame; here we are just animating one artist, the image, in each frame
ims = []
for i in range(60):
    x += np.pi / 15.
y += np.pi / 20.

#74.7. animation example code: dynamic_image2.py 1635
im = plt.imshow(f(x, y))
ims.append([im])

ani = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                               repeat_delay=1000)

#ani.save('dynamic_images.mp4')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.8 animation example code: histogram.py

[source code]

"""
This example shows how to use a path patch to draw a bunch of
rectangles for an animated histogram
"""

import numpy as np

import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path
import matplotlib.animation as animation

fig, ax = plt.subplots()

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
nrects = len(left)

# here comes the tricky part -- we have to set up the vertex and path
# codes arrays using moveto, lineto and closepoly

# for each rect: 1 for the MOVETO, 3 for the LINETO, 1 for the
# CLOSEPOLY; the vert for the closepoly is ignored but we still need
# it to keep the codes aligned with the vertices
nverts = nrects*5
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5,0] = left
verts[0::5,1] = bottom
verts[1::5,0] = left
verts[1::5,1] = top
verts[2::5,0] = right
verts[2::5,1] = top
verts[3::5,0] = right
verts[3::5,1] = bottom

barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

def animate(i):
    # simulate new data coming in
    data = np.random.randn(1000)
    n, bins = np.histogram(data, 100)
    top = bottom + n
    verts[1::5,1] = top
    verts[2::5,1] = top

ani = animation.FuncAnimation(fig, animate, 100, repeat=False)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.9 animation example code: moviewriter.py

[source code]

# This example uses a MovieWriter directly to grab individual frames and
# write them to a file. This avoids any event loop integration, but has
# the advantage of working with even the Agg backend. This is not recommended
# for use in an interactive setting.
# -*- noplot -*-

import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import matplotlib.animation as manimation

FFMpegWriter = manimation.writers['ffmpeg']
metadata = dict(title='Movie Test', artist='Matplotlib',
                comment='Movie support!')
writer = FFMpegWriter(fps=15, metadata=metadata)
fig = plt.figure()
l, = plt.plot([], [], 'k-o')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
x0, y0 = 0, 0

with writer.saving(fig, "writer_test.mp4", 100):
    for i in range(100):
        x0 += 0.1 * np.random.randn()
y0 += 0.1 * np.random.randn()
l.set_data(x0, y0)
        writer.grab_frame()
# Construct the scatter which we will update during animation
# as the raindrops develop.
scat = ax.scatter(rain_drops['position'][:,0], rain_drops['position'][:,1],
                 s=rain_drops['size'], lw=0.5, edgecolors=rain_drops['color'],
                 facecolors='none')

def update(frame_number):
    # Get an index which we can use to re-spawn the oldest raindrop.
    current_index = frame_number % n_drops

    # Make all colors more transparent as time progresses.
    rain_drops['color'][:, 3] -= 1.0/len(rain_drops)
    rain_drops['color'][:,3] = np.clip(rain_drops['color'][:,3], 0, 1)

    # Make all circles bigger.
    rain_drops['size'] += rain_drops['growth']

    # Pick a new position for oldest rain drop, resetting its size,
    # color and growth factor.
    rain_drops['position'][current_index] = np.random.uniform(0, 1, 2)
    rain_drops['size'][current_index] = 5
    rain_drops['color'][current_index] = (0, 0, 0, 1)
    rain_drops['growth'][current_index] = np.random.uniform(50, 200)

    # Update the scatter collection, with the new colors, sizes and positions.
    scat.set_edgecolors(rain_drops['color'])
    scat.set_sizes(rain_drops['size'])
    scat.set_offsets(rain_drops['position'])

# Construct the animation, using the update function as the animation
director.
animation = FuncAnimation(fig, update, interval=10)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.11 animation example code: random_data.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig, ax = plt.subplots()
line, = ax.plot(np.random.rand(10))
ax.set_ylim(0, 1)
```
```python
def update(data):
    line.set_ydata(data)
    return line,
def data_gen():
    while True: yield np.random.rand(10)

ani = animation.FuncAnimation(fig, update, data_gen, interval=100)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.12 animation example code: simple_3danim.py

[source code]

"""
A simple example of an animated plot... In 3D!
"""
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation
def Gen_RandLine(length, dims=2):
    """
    Create a line using a random walk algorithm
    
    length is the number of points for the line.
    dims is the number of dimensions the line has.
    """
    lineData = np.empty((dims, length))
    lineData[:, 0] = np.random.rand(dims)
    for index in range(1, length):
        # scaling the random numbers by 0.1 so
        # movement is small compared to position.
        # subtraction by 0.5 is to change the range to [-0.5, 0.5]
        # to allow a line to move backwards.
        step = ((np.random.rand(dims) - 0.5) * 0.1)
        lineData[:, index] = lineData[:, index-1] + step
    return lineData
def update_lines(num, dataLines, lines):
    for line, data in zip(lines, dataLines):
        # NOTE: there is no .set_data() for 3 dim data...
        line.set_data(data[0:2, :num])
        line.set_3d_properties(data[2,:num])
    return lines

# Attaching 3D axis to the figure
fig = plt.figure()
ax = p3.Axes3D(fig)

# Fifty lines of random 3-D lines
data = [Gen_RandLine(25, 3) for index in range(50)]

# Creating fifty line objects.
# NOTE: Can't pass empty arrays into 3d version of plot()
lines = [ax.plot(dat[0, 0:1], dat[1, 0:1], dat[2, 0:1])[0] for dat in data]

# Setting the axes properties
ax.set_xlim3d([0.0, 1.0])
ax.set_xlabel('X')
ax.set_ylim3d([0.0, 1.0])
ax.set_ylabel('Y')
ax.set_zlim3d([0.0, 1.0])
ax.set_zlabel('Z')
ax.set_title('3D Test')

# Creating the Animation object
line_ani = animation.FuncAnimation(fig, update_lines, 25, fargs=(data, lines),
interval=50, blit=False)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.13 animation example code: simple_anim.py

[source code]

"""
A simple example of an animated plot
"""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig, ax = plt.subplots()

x = np.arange(0, 2*np.pi, 0.01)  # x-array
line, = ax.plot(x, np.sin(x))

def animate(i):
    line.set_ydata(np.sin(x+i/10.0))  # update the data
    return line,

#Init only required for blitting to give a clean slate.
**74.14 animation example code: strip_chart_demo.py**

[source code]

""
Emulate an oscilloscope. Requires the animation API introduced in matplotlib 1.0 SVN.
""

```python
import numpy as np
from matplotlib.lines import Line2D
import matplotlib.pyplot as plt
import matplotlib.animation as animation
class Scope:
    def __init__(self, ax, maxt=2, dt=0.02):
        self.ax = ax
        self.dt = dt
        self.maxt = maxt
        self.tdata = [0]
        self.ydata = [0]
        self.line = Line2D(self.tdata, self.ydata)
        self.ax.add_line(self.line)
        self.ax.set_ylim(-.1, 1.1)
        self.ax.set_xlim(0, self.maxt)
    def update(self, y):
        lastt = self.tdata[-1]
        if lastt > self.tdata[0] + self.maxt:  # reset the arrays
            self.tdata = [self.tdata[-1]]
            self.ydata = [self.ydata[-1]]
            self.ax.set_xlim(self.tdata[0], self.tdata[0] + self.maxt)
            self.ax.figure.canvas.draw()
        t = self.tdata[-1] + self.dt
        self.tdata.append(t)
        self.ydata.append(y)
        self.line.set_data(self.tdata, self.ydata)
        return self.line,
    def emitter(p=0.03):
```
'return a random value with probability p, else 0'

while True:
    v = np.random.rand(1)
    if v > p:
        yield 0.
    else:
        yield np.random.rand(1)

fig, ax = plt.subplots()
scope = Scope(ax)

# pass a generator in "emitter" to produce data for the update func
ani = animation.FuncAnimation(fig, scope.update, emitter, interval=10,
                               blit=True)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.15 animation example code: subplots.py

[source code]

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import matplotlib.animation as animation

# This example uses subclassing, but there is no reason that the proper function
# couldn't be set up and then use FuncAnimation. The code is long, but not
# really complex. The length is due solely to the fact that there are a total
# of 9 lines that need to be changed for the animation as well as 3 subplots
# that need initial set up.
class SubplotAnimation(animation.TimedAnimation):
    def __init__(self):
        fig = plt.figure()
        ax1 = fig.add_subplot(1, 2, 1)
        ax2 = fig.add_subplot(2, 2, 2)
        ax3 = fig.add_subplot(2, 2, 4)

        self.t = np.linspace(0, 80, 400)
        self.x = np.cos(2 * np.pi * self.t / 10.)
        self.y = np.sin(2 * np.pi * self.t / 10.)
        self.z = 10 * self.t

        ax1.set_xlabel('x')
        ax1.set_ylabel('y')
        self.line1 = Line2D([], [], color='black')
        self.line1a = Line2D([], [], color='red', linewidth=2)
        self.line1e = Line2D([], [], color='red', marker='o', markeredgecolor='r')
ax1.add_line(self.line1)
ax1.add_line(self.line1a)
ax1.add_line(self.line1e)
ax1.set_xlim(-1, 1)
ax1.set_ylim(-2, 2)
ax1.set_aspect('equal', 'datalim')

ax2.set_xlabel('y')
ax2.set_ylabel('z')
self.line2 = Line2D([], [], color='black')
self.line2a = Line2D([], [], color='red', linewidth=2)
self.line2e = Line2D([], [], color='red', marker='o', markeredgecolor='r')
ax2.add_line(self.line2)
ax2.add_line(self.line2a)
ax2.add_line(self.line2e)
ax2.set_xlabel(-1, 1)
ax2.set_ylim(0, 800)

ax3.set_xlabel('x')
ax3.set_ylabel('z')
self.line3 = Line2D([], [], color='black')
self.line3a = Line2D([], [], color='red', linewidth=2)
self.line3e = Line2D([], [], color='red', marker='o', markeredgecolor='r')
ax3.add_line(self.line3)
ax3.add_line(self.line3a)
ax3.add_line(self.line3e)
ax3.set_xlim(-1, 1)
ax3.set_ylim(0, 800)

animation.TimedAnimation.__init__(self, fig, interval=50, blit=True)

def _draw_frame(self, framedata):
i = framedata
head = i - 1
head_len = 10
head_slice = (self.t > self.t[i] - 1.0) & (self.t < self.t[i])

self.line1.set_data(self.x[:i], self.y[:i])
self.line1a.set_data(self.x[head_slice], self.y[head_slice])
self.line1e.set_data(self.x[head], self.y[head])

self.line2.set_data(self.y[:i], self.z[:i])
self.line2a.set_data(self.y[head_slice], self.z[head_slice])
self.line2e.set_data(self.y[head], self.z[head])

self.line3.set_data(self.x[:i], self.z[:i])
self.line3a.set_data(self.x[head_slice], self.z[head_slice])
self.line3e.set_data(self.x[head], self.z[head])

self._drawn_artists = [self.line1, self.line1a, self.line1e,
                       self.line2, self.line2a, self.line2e,
                       self.line3, self.line3a, self.line3e]
```python
def new_frame_seq(self):
    return iter(range(self.t.size))

def _init_draw(self):
    lines = [self.line1, self.line1a, self.line1e,
             self.line2, self.line2a, self.line2e,
             self.line3, self.line3a, self.line3e]
    for l in lines:
        l.set_data([], [])

ani = SubplotAnimation()
    #ani.save('test_sub.mp4')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

74.16 animation example code: unchained.py

[source code]

""
Comparative path demonstration of frequency from a fake signal of a pulsar.
(mostly known because of the cover for Joy Division’s Unknown Pleasures)

Author: Nicolas P. Rougier
""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Create new Figure with black background
fig = plt.figure(figsize=(8, 8), facecolor='black')

# Add a subplot with no frame
ax = plt.subplot(111, frameon=False)

# Generate random data
data = np.random.uniform(0, 1, (64, 75))
X = np.linspace(-1, 1, data.shape[-1])
G = 1.5 * np.exp(-4 * X * X)

# Generate line plots
lines = []
for i in range(len(data)):
    # Small reduction of the X extents to get a cheap perspective effect
    xscale = 1 - i / 200.
    # Same for linewidth (thicker strokes on bottom)
    lw = 1.5 - i / 100.0
    line, = ax.plot(xscale * X, i + G * data[i], color="w", lw=lw)
    lines.append(line)

"""
# Set y limit (or first line is cropped because of thickness)
ax.set_ylim(-1, 70)

# No ticks
ax.set_xticks([])
ax.set_yticks([])

# 2 part titles to get different font weights
ax.text(0.5, 1.0, "MATPLOTLIB ", transform=ax.transAxes,
    ha="right", va="bottom", color="w",
    family="sans-serif", fontweight="light", fontsize=16)
ax.text(0.5, 1.0, "UNCHAINED", transform=ax.transAxes,
    ha="left", va="bottom", color="w",
    family="sans-serif", fontweight="bold", fontsize=16)

# Update function
def update(*args):
    # Shift all data to the right
data[:, 1:] = data[:, :-1]

    # Fill-in new values
    data[:, 0] = np.random.uniform(0, 1, len(data))

    # Update data
    for i in range(len(data)):
        lines[i].set_ydata(i + G * data[i])

    # Return modified artists
    return lines

# Construct the animation, using the update function as the animation
# director.
anim = animation.FuncAnimation(fig, update, interval=10)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
75.1 api example code: agg_oo.py

[source code]

#!/usr/bin/env python
# -*- noplot -*-

"""
A pure OO (look Ma, no pylab!) example using the agg backend
"""

from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.plot([1, 2, 3])
ax.set_title('hi mom')
ax.grid(True)
ax.set_xlabel('time')
ax.set_ylabel('volts')
canvas.print_figure('test')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
# a bar plot with errorbars
import numpy as np
import matplotlib.pyplot as plt

N = 5
menMeans = (20, 35, 30, 35, 27)
mensStd = (2, 3, 4, 1, 2)

ind = np.arange(N)  # the x locations for the groups
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(ind, menMeans, width, color='r', yerr=menStd)

womenMeans = (25, 32, 34, 20, 25)
womenStd = (3, 5, 2, 3, 3)
rects2 = ax.bar(ind + width, womenMeans, width, color='y', yerr=womenStd)

# add some text for labels, title and axes ticks
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
ax.set_xticks(ind + width / 2)
ax.set_xticklabels(('G1', 'G2', 'G3', 'G4', 'G5'))
ax.legend((rects1[0], rects2[0]), ('Men', 'Women'))

fig.tight_layout()
plt.show()
```python
ax.set_xticks(ind+width)
ax.set_xticklabels( ('G1', 'G2', 'G3', 'G4', 'G5') )

ax.legend( (rects1[0], rects2[0]), ('Men', 'Women') )

def autolabel(rects):
    # attach some text labels
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x()+rect.get_width()/2., 1.05*height, '%d'%int(height),
                ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.3 api example code: bbox_intersect.py
```

---

75.3. api example code: bbox_intersect.py 1649
Matplotlib, Release 1.4.3

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.transforms import Bbox
from matplotlib.path import Path
rect = plt.Rectangle((-1, -1), 2, 2, facecolor="#aaaaaa")
plt.gca().add_patch(rect)
bbox = Bbox.from_bounds(-1, -1, 2, 2)
for i in range(12):
vertices = (np.random.random((2, 2)) - 0.5) * 6.0
path = Path(vertices)
if path.intersects_bbox(bbox):
color = 'r'
else:
color = 'b'
plt.plot(vertices[:,0], vertices[:,1], color=color)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.4 api example code: collections_demo.py

LineCollection
using offsets
PolyCollection
using offsets
3
3

Depth (m)

2
2
1
1
0
0
1
1
2
2
3
3
3
2
1
0
1
2
3
3 2 1 0data
1 offsets
2 3
RegularPolyCollection
using
offsets
Successive
3
0
100
2
200
1
300
0
400
1
500
2
600
3
700
3 2 1 0 1 2 3 0.50.00.51.01.52.02.53.0
Zonal velocity component (m/s)
1650

Chapter 75. api Examples


#!/usr/bin/env python
"Demonstration of LineCollection, PolyCollection, and
RegularPolyCollection with autoscaling.

For the first two subplots, we will use spirals. Their
size will be set in plot units, not data units. Their positions
will be set in data units by using the "offsets" and "transOffset"
kwags of the LineCollection and PolyCollection.

The third subplot will make regular polygons, with the same
type of scaling and positioning as in the first two.

The last subplot illustrates the use of "offsets=(xo,yo)",
that is, a single tuple instead of a list of tuples, to generate
successively offset curves, with the offset given in data
units. This behavior is available only for the LineCollection.

"

import matplotlib.pyplot as plt
from matplotlib import collections, transforms
from matplotlib.colors import colorConverter
import numpy as np

nverts = 50
npts = 100

# Make some spirals
r = np.array(range(nverts))
theta = np.array(range(nverts)) * (2*np.pi)/(nverts-1)
xx = r * np.sin(theta)
yy = r * np.cos(theta)
spiral = list(zip(xx,yy))

# Make some offsets
rs = np.random.RandomState([12345678])
oxo = rs.randn(npts)
ayo = rs.randn(npts)
xyo = list(zip(xo, yo))

# Make a list of colors cycling through the rgbcmyk series.
colors = [colorConverter.to_rgba(c) for c in ('r','g','b','c','y','m','k')]

fig, axes = plt.subplots(2,2)
((ax1, ax2), (ax3, ax4)) = axes # unpack the axes

col = collections.LineCollection([spiral], offsets=xyo,
transOffset=ax1.transData)
trans = fig.dpi_scale_trans + transforms.Affine2D().scale(1.0/72.0)
col.set_transform(trans) # the points to pixels transform
    # Note: the first argument to the collection initializer
    # must be a list of sequences of x,y tuples; we have only
# one sequence, but we still have to put it in a list.
ax1.add_collection(col, autolim=True)
# autolim=True enables autoscaling. For collections with
# offsets like this, it is neither efficient nor accurate,
# but it is good enough to generate a plot that you can use
# as a starting point. If you know beforehand the range of
# x and y that you want to show, it is better to set them
# explicitly, leave out the autolim kwarg (or set it to False),
# and omit the 'ax1.autoscale_view()' call below.

# Make a transform for the line segments such that their size is
# given in points:
col.set_color(colors)
ax1.autoscale_view()  # See comment above, after ax1.add_collection.
ax1.set_title('LineCollection using offsets')

# The same data as above, but fill the curves.
col = collections.PolyCollection([spiral], offsets=xyo,
                                  transOffset=ax2.transData)
trans = transforms.Affine2D().scale(fig.dpi/72.0)
col.set_transform(trans)  # the points to pixels transform
ax2.add_collection(col, autolim=True)
col.set_color(colors)

ax2.autoscale_view()
ax2.set_title('PolyCollection using offsets')

# 7-sided regular polygons

col = collections.RegularPolyCollection(7,
                                  sizes = np.fabs(xx)*10.0, offsets=xyo,
                                  transOffset=ax3.transData)
trans = transforms.Affine2D().scale(fig.dpi/72.0)
col.set_transform(trans)  # the points to pixels transform
ax3.add_collection(col, autolim=True)
col.set_color(colors)
ax3.autoscale_view()
ax3.set_title('RegularPolyCollection using offsets')

# Simulate a series of ocean current profiles, successively
# offset by 0.1 m/s so that they form what is sometimes called
# a "waterfall" plot or a "stagger" plot.
nverts = 60
ncurves = 20
offs = (0.1, 0.0)

yy = np.linspace(0, 2*np.pi, nverts)
ym = np.amax(yy)
Matplotlib, Release 1.4.3

```
xx = (0.2 + (ym-yy)/ym)**2 * np.cos(yy-0.4) * 0.5
segs = []
for i in range(ncurves):
    xxx = xx + 0.02*rs.randn(nverts)
    curve = list(zip(xxx, yy*100))
    segs.append(curve)

col = collections.LineCollection(segs, offsets=offs)
ax4.add_collection(col, autolim=True)
col.set_color(colors)
ax4.autoscale_view()
ax4.set_title('Successive data offsets')
ax4.set_xlabel('Zonal velocity component (m/s)')
ax4.set_ylabel('Depth (m)')
# Reverse the y-axis so depth increases downward
ax4.set_ylim(ax4.get_ylim()[::-1])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.5 api example code: colorbar_only.py

```
from matplotlib import pyplot
import matplotlib as mpl
# Make a figure and axes with dimensions as desired.
fig = pyplot.figure(figsize=(8,3))
ax1 = fig.add_axes([0.05, 0.80, 0.9, 0.15])
ax2 = fig.add_axes([0.05, 0.475, 0.9, 0.15])
```

75.5. api example code: colorbar_only.py
ax3 = fig.add_axes([0.05, 0.15, 0.9, 0.15])

# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
cmap = mpl.cm.cool
norm = mpl.colors.Normalize(vmin=5, vmax=10)

# ColorbarBase derives from ScalarMappable and puts a colorbar
# in a specified axes, so it has everything needed for a
# standalone colorbar. There are many more kwargs, but the
# following gives a basic continuous colorbar with ticks
# and labels.
{cb1 = mpl.colorbar.ColorbarBase(ax1, cmap=cmap,
                             norm=norm,
                             orientation='horizontal')

cb1.set_label('Some Units')

# The second example illustrates the use of a ListedColormap, a
# BoundaryNorm, and extended ends to show the "over" and "under"
# value colors.
cmap = mpl.colors.ListedColormap(['r', 'g', 'b', 'c'])
cmap.set_over('0.25')
cmap.set_under('0.75')

# If a ListedColormap is used, the length of the bounds array must be
# one greater than the length of the color list. The bounds must be
# monotonically increasing.
bounds = [1, 2, 4, 7, 8]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
{cb2 = mpl.colorbar.ColorbarBase(ax2, cmap=cmap,
                             norm=norm,
                             boundaries=[0]+bounds+[13],
                             extend='both',
                             ticks=bounds, # optional
                             spacing='proportional',
                             orientation='horizontal')

cb2.set_label('Discrete intervals, some other units')

# The third example illustrates the use of custom length colorbar
# extensions, used on a colorbar with discrete intervals.
cmap = mpl.colors.ListedColormap([[0., .4, 1.], [0., .8, 1.],
                                    [1., .8, 0.], [1., .4, 0.]])
cmap.set_over((1., 0., 0.))
cmap.set_under((0., 0., 1.))

bounds = [-1., -.5, 0., .5, 1.]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
{cb3 = mpl.colorbar.ColorbarBase(ax3, cmap=cmap,
                             norm=norm,
                             boundaries=[-10]+bounds+[10],
                             extend='both')}
# Make the length of each extension
# the same as the length of the
# interior colors:
extendfrac='auto',
ticks=bounds,
spacing='uniform',
orientation='horizontal')

cb3.set_label('Custom extension lengths, some other units')

pyplot.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

## 75.6 api example code: compound_path.py

```python
import numpy as np
from matplotlib.path import Path

"""
Make a compound path -- in this case two simple polygons, a rectangle and a triangle. Use CLOSEPOLY and MOVETO for the different parts of the compound path
"""

A compound path

75.6. api example code: compound_path.py

1655
from matplotlib.patches import PathPatch
import matplotlib.pyplot as plt

vertices = []
codes = []

codes = [Path.MOVETO] + [Path.LINETO]*3 + [Path.CLOSEPOLY]
vertices = [(1,1), (1,2), (2, 2), (2, 1), (0,0)]

codes += [Path.MOVETO] + [Path.LINETO]*2 + [Path.CLOSEPOLY]
vertices += [(4,4), (5,5), (5, 4), (0,0)]

vertices = np.array(vertices, float)
path = Path(vertices, codes)

pathpatch = PathPatch(path, facecolor='None', edgecolor='green')

fig, ax = plt.subplots()
ax.add_patch(pathpatch)
ax.set_title('A compound path')
ax.dataLim.update_from_data_xy(vertices)
ax.autoscale_view()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from __future__ import unicode_literals

import matplotlib
from matplotlib.axes import Axes
from matplotlib.patches import Circle
from matplotlib.path import Path
from matplotlib.ticker import NullLocator, Formatter, FixedLocator
from matplotlib.transforms import Affine2D, BboxTransformTo, Transform
from matplotlib.projections import register_projection
import matplotlib.spines as mspines
import matplotlib.axis as maxis
import numpy as np

# This example projection class is rather long, but it is designed to
# illustrate many features, not all of which will be used every time.
# It is also common to factor out a lot of these methods into common
# code used by a number of projections with similar characteristics
# (see geo.py).

class HammerAxes(Axes):
    
75.7 api example code: custom_projection_example.py
A custom class for the Aitoff-Hammer projection, an equal-area map projection.


# The projection must specify a name. This will be used by the
# user to select the projection, i.e. ``subplot(111,
# projection='custom_hammer')``.
name = 'custom_hammer'

def __init__(self, *args, **kwargs):
    Axes.__init__(self, *args, **kwargs)
    self.set_aspect(0.5, adjustable='box', anchor='C')
    self.cla()

def _init_axis(self):
    self.xaxis = maxis.XAxis(self)
    self.yaxis = maxis.YAxis(self)
    # Do not register xaxis or yaxis with spines -- as done in
    # Axes._init_axis() -- until HammerAxes.xaxis.cla() works.
    # self.spines['hammer'].register_axis(self.yaxis)
    self._update_transScale()

def cla(self):
    ""
    Override to set up some reasonable defaults.
    ""
    # Don't forget to call the base class
    Axes.cla(self)

    # Set up a default grid spacing
    self.set_longitude_grid(30)
    self.set_latitude_grid(15)
    self.set_longitude_grid_ends(75)

    # Turn off minor ticking altogether
    self.xaxis.set_minor_locator(NullLocator())
    self.yaxis.set_minor_locator(NullLocator())

    # Do not display ticks -- we only want gridlines and text
    self.xaxis.set_ticks_position('none')
    self.yaxis.set_ticks_position('none')

    # The limits on this projection are fixed -- they are not to
    # be changed by the user. This makes the math in the
    # transformation itself easier, and since this is a toy
    # example, the easier, the better.
    Axes.set_xlim(self, -np.pi, np.pi)
    Axes.set_ylim(self, -np.pi / 2.0, np.pi / 2.0)

def _set_lim_and_transforms(self):
    ""
    This is called once when the plot is created to set up all the
transforms for the data, text and grids.

There are three important coordinate spaces going on here:

1. Data space: The space of the data itself
2. Axes space: The unit rectangle (0, 0) to (1, 1) covering the entire plot area.
3. Display space: The coordinates of the resulting image, often in pixels or dpi/inch.

This function makes heavy use of the Transform classes in `lib/matplotlib/transforms.py`. For more information, see the inline documentation there.

The goal of the first two transformations is to get from the data space (in this case longitude and latitude) to axes space. It is separated into a non-affine and affine part so that the non-affine part does not have to be recomputed when a simple affine change to the figure has been made (such as resizing the window or changing the dpi).

1) The core transformation from data space into rectilinear space defined in the HammerTransform class.

```python
self.transProjection = self.HammerTransform()
```

2) The above has an output range that is not in the unit rectangle, so scale and translate it so it fits correctly within the axes. The peculiar calculations of xscale and yscale are specific to a Aitoff-Hammer projection, so don't worry about them too much.

```python
xscale = 2.0 * np.sqrt(2.0) * np.sin(0.5 * np.pi)
yscale = np.sqrt(2.0) * np.sin(0.5 * np.pi)
self.transAffine = Affine2D() \ .scale(0.5 / xscale, 0.5 / yscale) \ .translate(0.5, 0.5)
```

3) This is the transformation from axes space to display space.

```python
self.transAxes = BboxTransformTo(self.bbox)
```

Now put these 3 transforms together -- from data all the way to display coordinates. Using the `+` operator, these transforms will be applied "in order". The transforms are automatically simplified, if possible, by the underlying transformation framework.

```python
self.transData = \ self.transProjection + \ self.transAffine + \ self.transAxes
```

The main data transformation is set up. Now deal with
# gridlines and tick labels.

# Longitude gridlines and ticklabels. The input to these
# transforms are in display space in x and axes space in y.
# Therefore, the input values will be in range (-xmin, 0),
# (xmax, 1). The goal of these transforms is to go from that
# space to display space. The tick labels will be offset 4
# pixels from the equator.
self._xaxis_pretransform = \
    Affine2D() \
    .scale(1.0, np.pi) \
    .translate(0.0, -np.pi)
self._xaxis_transform = \
    self._xaxis_pretransform + \
    self.transData
self._xaxis_text1_transform = \
    Affine2D().scale(1.0, 0.0) + \
    self.transData + \
    Affine2D().translate(0.0, 4.0)
self._xaxis_text2_transform = \
    Affine2D().scale(1.0, 0.0) + \
    self.transData + \
    Affine2D().translate(0.0, -4.0)

# Now set up the transforms for the latitude ticks. The input to
# these transforms are in axes space in x and display space in
# y. Therefore, the input values will be in range (0, -ymin),
# (1, ymax). The goal of these transforms is to go from that
# space to display space. The tick labels will be offset 4
# pixels from the edge of the axes ellipse.
yaxis_stretch = Affine2D().scale(np.pi * 2.0, 1.0).translate(-np.pi, 0.0)
yaxis_space = Affine2D().scale(1.0, 1.1)
self._yaxis_transform = \
    yaxis_stretch + \
    self.transData
yaxis_text_base = \
    yaxis_stretch + \
    self.transProjection + \
    (yaxis_space + \
     self.transAffine + \
     self.transAxes)
self._yaxis_text1_transform = \
    yaxis_text_base + \
    Affine2D().translate(-8.0, 0.0)
self._yaxis_text2_transform = \
    yaxis_text_base + \
    Affine2D().translate(8.0, 0.0)

def get_xaxis_transform(self, which='grid'):
    """
    Override this method to provide a transformation for the
    x-axis grid and ticks.
    """
```
assert which in ['tick1','tick2','grid']
return self._xaxis_transform

def get_xaxis_text1_transform(self, pixelPad):
    ""
    Override this method to provide a transformation for the
    x-axis tick labels.
    ""
    return self._xaxis_text1_transform, 'bottom', 'center'

def get_xaxis_text2_transform(self, pixelPad):
    ""
    Override this method to provide a transformation for the
    secondary x-axis tick labels.
    ""
    return self._xaxis_text2_transform, 'top', 'center'

def get_yaxis_transform(self, which='grid'):
    ""
    Override this method to provide a transformation for the
    y-axis grid and ticks.
    ""
    assert which in ['tick1','tick2','grid']
    return self._yaxis_transform

def get_yaxis_text1_transform(self, pixelPad):
    ""
    Override this method to provide a transformation for the
    y-axis tick labels.
    ""
    return self._yaxis_text1_transform, 'center', 'right'

def get_yaxis_text2_transform(self, pixelPad):
    ""
    Override this method to provide a transformation for the
    secondary y-axis tick labels.
    ""
    return self._yaxis_text2_transform, 'center', 'left'

def _gen_axes_patch(self):
    ""
    Override this method to define the shape that is used for the
    background of the plot. It should be a subclass of Patch.
    ""
    In this case, it is a Circle (that may be warped by the axes
```
transform into an ellipse). Any data and gridlines will be clipped to this shape.

```
return Circle((0.5, 0.5), 0.5)
```

```python
def _gen_axes_spines(self):
    return {'custom_hammer': mspines.Spine.circular_spine(self,
        (0.5, 0.5), 0.5)}
```

```python
# Prevent the user from applying scales to one or both of the
# axes. In this particular case, scaling the axes wouldn't make
# sense, so we don't allow it.
def set_xscale(self, *args, **kwargs):
    if args[0] != 'linear':
        raise NotImplementedError
    Axes.set_xscale(self, *args, **kwargs)

def set_yscale(self, *args, **kwargs):
    if args[0] != 'linear':
        raise NotImplementedError
    Axes.set_yscale(self, *args, **kwargs)
```

```python
# Prevent the user from changing the axes limits. In our case, we
# want to display the whole sphere all the time, so we override
# set_xlim and set_ylim to ignore any input. This also applies to
# interactive panning and zooming in the GUI interfaces.
def set_xlim(self, *args, **kwargs):
    Axes.set_xlim(self, -np.pi, np.pi)
    Axes.set_ylim(self, -np.pi / 2.0, np.pi / 2.0)
    set_ylim = set_xlim

def format_coord(self, lon, lat):
    """
    Override this method to change how the values are displayed in
    the status bar.

    In this case, we want them to be displayed in degrees N/S/E/W.
    """
    lon = lon * (180.0 / np.pi)
    lat = lat * (180.0 / np.pi)
    if lat >= 0.0:
        ns = 'N'
    else:
        ns = 'S'
    if lon >= 0.0:
        ew = 'E'
    else:
        ew = 'W'
        # \u00b0 : degree symbol
    return '%s\u00b0s, %s\u00b0s' % (abs(lat), ns, abs(lon), ew)
```

```python
class DegreeFormatter(Formatter):
    """
```
This is a custom formatter that converts the native unit of radians into (truncated) degrees and adds a degree symbol.

```python
def __init__(self, round_to=1.0):
    self._round_to = round_to

def __call__(self, x, pos=None):
    degrees = (x / np.pi) * 180.0
    degrees = round(degrees / self._round_to) * self._round_to
    # ° : degree symbol
    return "\%d\u00b0" % degrees

def set_longitude_grid(self, degrees):
    ""
    Set the number of degrees between each longitude grid.
    This is an example method that is specific to this projection class -- it provides a more convenient interface to set the ticking than set_xticks would.
    ""
    # Set up a FixedLocator at each of the points, evenly spaced # by degrees.
    number = (360.0 / degrees) + 1
    self.xaxis.set_major_locator(
        plt.FixedLocator(
            np.linspace(-np.pi, np.pi, number, True)[1:-1]))
    # Set the formatter to display the tick labels in degrees,
    # rather than radians.
    self.xaxis.set_major_formatter(self.DegreeFormatter(degrees))

def set_latitude_grid(self, degrees):
    ""
    Set the number of degrees between each longitude grid.
    This is an example method that is specific to this projection class -- it provides a more convenient interface than set_yticks would.
    ""
    # Set up a FixedLocator at each of the points, evenly spaced # by degrees.
    number = (180.0 / degrees) + 1
    self.yaxis.set_major_locator(
        FixedLocator(
            np.linspace(-np.pi / 2.0, np.pi / 2.0, number, True)[1:-1]))
    # Set the formatter to display the tick labels in degrees,
    # rather than radians.
    self.yaxis.set_major_formatter(self.DegreeFormatter(degrees))

def set_longitude_grid_ends(self, degrees):
    ""
    Set the latitude(s) at which to stop drawing the longitude grids.
    Often, in geographic projections, you wouldn't want to draw
longitude gridlines near the poles. This allows the user to specify the degree at which to stop drawing longitude grids.

This is an example method that is specific to this projection class -- it provides an interface to something that has no analogy in the base Axes class.

```python
longitude_cap = degrees * (np.pi / 180.0)
# Change the xaxis gridlines transform so that it draws from # -degrees to degrees, rather than -pi to pi.
self._xaxis_pretransform
  .clear()
  .scale(1.0, longitude_cap * 2.0)
  .translate(0.0, -longitude_cap)
```

def get_data_ratio(self):
    ""
    Return the aspect ratio of the data itself.
    ""
    return 1.0

# Interactive panning and zooming is not supported with this projection,
# so we override all of the following methods to disable it.

def can_zoom(self):
    ""
    Return True if this axes support the zoom box
    ""
    return False

def start_pan(self, x, y, button):
    pass

def end_pan(self):
    pass

def drag_pan(self, button, key, x, y):
    pass

# Now, the transforms themselves.

class HammerTransform(Transform):
    ""
    The base Hammer transform.
    ""
    input_dims = 2
    output_dims = 2
    is_separable = False

def transform_non_affine(self, ll):
    ""
    Override the transform_non_affine method to implement the custom transform.
The input and output are Nx2 numpy arrays.

```python
longitude = ll[:, 0:1]
latitude = ll[:, 1:2]

# Pre-compute some values
half_long = longitude / 2.0
cos_latitude = np.cos(latitude)
sqrt2 = np.sqrt(2.0)

alpha = 1.0 + cos_latitude * np.cos(half_long)
x = (2.0 * sqrt2) * (cos_latitude * np.sin(half_long)) / alpha
y = (sqrt2 * np.sin(latitude)) / alpha
return np.concatenate((x, y), 1)
```

This is where things get interesting. With this projection, straight lines in data space become curves in display space. This is done by interpolating new values between the input values of the data. Since ``transform`` must not return a differently-sized array, any transform that requires changing the length of the data array must happen within ``transform_path``.

```python
def transform_path_non_affine(self, path):
ipath = path.interpolated(path._interpolation_steps)
return Path(self.transform(ipath.vertices), ipath.codes)
```

```python
transform_path_non_affine.__doc__ = Transform.transform_path_non_affine.__doc__
```

```python
if matplotlib.__version__ < '1.2':
    # Note: For compatibility with matplotlib v1.1 and older, you'll need to explicitly implement a ``transform`` method as well.
    # Otherwise a `NotImplementedError` will be raised. This isn't necessary for v1.2 and newer, however.
    transform = transform_non_affine

    # Similarly, we need to explicitly override ``transform_path`` if compatibility with older matplotlib versions is needed. With v1.2 and newer, only overriding the ``transform_path_non_affine`` method is sufficient.
    transform_path = transform_path_non_affine
    transform_path.__doc__ = Transform.transform_path.__doc__
```

```python
def inverted(self):
    return HammerAxes.InvertedHammerTransform()
inverted.__doc__ = Transform.inverted.__doc__
```

```python
class InvertedHammerTransform(Transform):
    input_dims = 2
    output_dims = 2
    is_separable = False

    def transform_non_affine(self, xy):
        x = xy[:, 0:1]
```

75.7. api example code: custom_projection_example.py
y = xy[:, 1:2]

quarter_x = 0.25 * x
half_y = 0.5 * y
z = np.sqrt(1.0 - quarter_x*quarter_x - half_y*half_y)
longitude = 2 * np.arctan((z*x) / (2.0 * (2.0*z*z - 1.0)))
latitude = np.arcsin(y*z)

return np.concatenate((longitude, latitude), 1)

transform_non_affine.__doc__ = Transform.transform_non_affine.__doc__

# As before, we need to implement the "transform" method for 
# compatibility with matplotlib v1.1 and older.
if matplotlib.__version__ < '1.2':
    transform = transform_non_affine

def inverted(self):
    # The inverse of the inverse is the original transform... ;)
    return HammerAxes.HammerTransform()

inverted.__doc__ = Transform.inverted.__doc__

# Now register the projection with matplotlib so the user can select 
# it.
register_projection(HammerAxes)

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    # Now make a simple example using the custom projection.
    plt.subplot(111, projection="custom_hammer")
    p = plt.plot([-1, 1, 1], [-1, -1, 1], "o-")
    plt.grid(True)

    plt.show()
from __future__ import unicode_literals

import numpy as np
from numpy import ma
from matplotlib import scale as mscale
from matplotlib import transforms as mtransforms
from matplotlib.ticker import Formatter, FixedLocator

class MercatorLatitudeScale(mscale.ScaleBase):
    """
    Scales data in range -pi/2 to pi/2 (-90 to 90 degrees) using the system used to scale latitudes in a Mercator projection.
    """

    The scale function:
    \( \ln(\tan(y) + \sec(y)) \)

    The inverse scale function:
    \( \arctan(\sinh(y)) \)

    Since the Mercator scale tends to infinity at +/- 90 degrees, there is user-defined threshold, above and below which nothing
will be plotted. This defaults to +/- 85 degrees.

source:
http://en.wikipedia.org/wiki/Mercator_projection

```python
# The scale class must have a member `name` that defines the string used to select the scale. For example, # `gca().set_yscale("mercator")` would be used to select this scale.
name = 'mercator'

def __init__(self, axis, **kwargs):
    
    Any keyword arguments passed to `set_xscale` and `set_yscale` will be passed along to the scale's constructor.

    thresh: The degree above which to crop the data.

def get_transform(self):
    
    Override this method to return a new instance that does the actual transformation of the data.

    The MercatorLatitudeTransform class is defined below as a nested class of this one.

    return self.MercatorLatitudeTransform(self.thresh)

def set_default_locators_and_formatters(self, axis):
    
    Override to set up the locators and formatters to use with the scale. This is only required if the scale requires custom locators and formatters. Writing custom locators and formatters is rather outside the scope of this example, but there are many helpful examples in `ticker.py`.

    In our case, the Mercator example uses a fixed locator from -90 to 90 degrees and a custom formatter class to put convert the radians to degrees and put a degree symbol after the value:

class DegreeFormatter(Formatter):
    def __call__(self, x, pos=None):
        # \u00b0 : degree symbol
```
```python
return "\%d\u00b0" % ((x / np.pi) * 180.0)

deg2rad = np.pi / 180.0
axis.set_major_locator(FixedLocator(
    np.arange(-90, 90, 10) * deg2rad))
axis.set_major_formatter(DegreeFormatter())
axis.set_minor_formatter(DegreeFormatter())

def limit_range_for_scale(self, vmin, vmax, minpos):
    
    Override to limit the bounds of the axis to the domain of the
    transform. In the case of Mercator, the bounds should be
    limited to the threshold that was passed in. Unlike the
    autoscaling provided by the tick locators, this range limiting
    will always be adhered to, whether the axis range is set
    manually, determined automatically or changed through panning
    and zooming.
    
    return max(vmin, -self.thresh), min(vmax, self.thresh)

class MercatorLatitudeTransform(mtransforms.Transform):
    # There are two value members that must be defined.
    # `input_dims` and `output_dims` specify number of input
    # dimensions and output dimensions to the transformation.
    # These are used by the transformation framework to do some
    # error checking and prevent incompatible transformations from
    # being connected together. When defining transforms for a
    # scale, which are, by definition, separable and have only one
    # dimension, these members should always be set to 1.
    input_dims = 1
    output_dims = 1
    is_separable = True

    def __init__(self, thresh):
        mtransforms.Transform.__init__(self)
        self.thresh = thresh

    def transform_non_affine(self, a):
        
        This transform takes an N\times 1 `numpy` array and returns a
        transformed copy. Since the range of the Mercator scale
        is limited by the user-specified threshold, the input
        array must be masked to contain only valid values.
        `matplotlib` will handle masked arrays and remove the
        out-of-range data from the plot. Importantly, the
        `transform` method *must* return an array that is the
        same shape as the input array, since these values need to
        remain synchronized with values in the other dimension.
        
        masked = ma.masked_where((a < -self.thresh) | (a > self.thresh), a)
        if masked.mask.any():
            return ma.log(np.abs(ma.tan(masked) + 1.0 / ma.cos(masked)))
        else:
```
return np.log(np.abs(np.tan(a) + 1.0 / np.cos(a)))

def inverted(self):
    """
    Override this method so matplotlib knows how to get the
    inverse transform for this transform.
    """
    return MercatorLatitudeScale.InvertedMercatorLatitudeTransform(self.thresh)

class InvertedMercatorLatitudeTransform(mtransforms.Transform):
    input_dims = 1
    output_dims = 1
    is_separable = True

def __init__(self, thresh):
    mtransforms.Transform.__init__(self)
    self.thresh = thresh

def transform_non_affine(self, a):
    return np.arctan(np.sinh(a))

def inverted(self):
    return MercatorLatitudeScale.MercatorLatitudeTransform(self.thresh)

# Now that the Scale class has been defined, it must be registered so
# that ``matplotlib`` can find it.
mscale.register_scale(MercatorLatitudeScale)

if __name__ == '__main__':
    import matplotlib.pyplot as plt

    t = np.arange(-180.0, 180.0, 0.1)
    s = t / 360.0 * np.pi

    plt.plot(t, s, '- ', lw=2)
    plt.gca().set_yscale('mercator')

    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.title('Mercator: Projection of the Oppressor')
    plt.grid(True)

    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

Show how to make date plots in matplotlib using date tick locators and formatters. See major_minor_demo1.py for more information on controlling major and minor ticks

All matplotlib date plotting is done by converting date instances into days since the 0001-01-01 UTC. The conversion, tick locating and formatting is done behind the scenes so this is most transparent to you. The dates module provides several converter functions date2num and num2date

```python
import datetime
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import matplotlib.cbook as cbook

years = mdates.YearLocator()    # every year
months = mdates.MonthLocator()  # every month
yearsFmt = mdates.DateFormatter('%Y')
```
# load a numpy record array from yahoo csv data with fields date,
# open, close, volume, adj_close from the mpl-data/example directory.
# The record array stores python datetime.date as an object array in
# the date column
datafile = cbook.get_sample_data('goog.npy')
r = np.load(datafile).view(np.recarray)

fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close)

# format the ticks
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(yearsFmt)
ax.xaxis.set_minor_locator(months)
datemin = datetime.date(r.date.min().year, 1, 1)
datemax = datetime.date(r.date.max().year+1, 1, 1)
ax.set_xlim(datemin, datemax)

# format the coords message box
def price(x):
    return '$%1.2f' % x
ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
ax.format_ydata = price
ax.grid(True)

# rotates and right aligns the x labels, and moves the bottom of the
# axes up to make room for them
fig.autofmt_xdate()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
75.10 api example code: date_index_formatter.py
When plotting time series, e.g., financial time series, one often wants to leave out days on which there is no data, eh weekends. The example below shows how to use an 'index formatter' to achieve the desired plot.

```python
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import matplotlib.ticker as ticker

datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading %s' % datafile)
r = mlab.csv2rec(datafile)

r.sort()
r = r[-30:]  # get the last 30 days

# first we'll do it the default way, with gaps on weekends
fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close, 'o-')
fig.autofmt_xdate()
```

"""
# next we'll write a custom formatter
N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices

def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y-%m-%d')

fig, ax = plt.subplots()
ax.plot(ind, r.adj_close, 'o-')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
fig.autofmt_xdate()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.11 api example code: demo_affine_image.py

#!/usr/bin/env python

75.11. api example code: demo_affine_image.py
For the backends that supports draw_image with optional affine transform (e.g., agg, ps backend), the image of the output should have its boundary matches the red rectangles.

```python
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

def get_image():
    delta = 0.25
    x = y = np.arange(-3.0, 3.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
    Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
    Z = Z2-Z1  # difference of Gaussians
    return Z

def imshow_affine(ax, z, *kl, **kwargs):
    im = ax.imshow(z, *kl, **kwargs)
    x1, x2, y1, y2 = im.get_extent()
    im._image_skew_coordinate = (x2, y1)
    return im

if 1:
    # image rotation
    fig, (ax1, ax2) = plt.subplots(1,2)
    Z = get_image()
    im1 = imshow_affine(ax1, Z, interpolation='none', cmap=cm.jet,
                        origin='lower',
                        extent=[-2, 4, -3, 2], clip_on=True)

    trans_data2 = mtransforms.Affine2D().rotate_deg(30) + ax1.transData
    im1.set_transform(trans_data2)

    # display intended extent of the image
    x1, x2, y1, y2 = im1.get_extent()
    x3, y3 = x2, y1

    ax1.plot([x1, x2, x2, x1, x1], [y1, y1, y2, y2, y1], "r--", lw=3,
             transform=trans_data2)

    ax1.set_xlim(-3, 5)
    ax1.set_ylim(-4, 4)
```

# image skew
im2 = ax2.imshow(Z, interpolation='none', cmap=cm.jet,
                      origin='lower',
                      extent=[-2, 4, -3, 2], clip_on=True)
im2._image_skew_coordinate = (3, -2)

plt.show()
# plt.savefig("demo_affine_image")

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.12 api example code: donut_demo.py

---

```python
import numpy as np
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

def wise(v):
    if v == 1:
        return "CCW"
    else:
        return "CW"
```

---

Mmm, donuts!

Outside CCW, Inside CCW
Outside CW, Inside CCW
Outside CCW, Inside CCW
Outside CW, Inside CCW
return "CW"

def make_circle(r):
    t = np.arange(0, np.pi * 2.0, 0.01)
    t = t.reshape((len(t), 1))
    x = r * np.cos(t)
    y = r * np.sin(t)
    return np.hstack((x, y))

Path = mpath.Path

fig, ax = plt.subplots()

inside_vertices = make_circle(0.5)
outside_vertices = make_circle(1.0)
codes = np.ones(len(inside_vertices), dtype=mpath.Path.code_type) * mpath.Path.LINETO
codes[0] = mpath.Path.MOVETO

for i, (inside, outside) in enumerate(((1, 1), (1, -1), (-1, 1), (-1, -1))):
    # Concatenate the inside and outside subpaths together, changing their
    # order as needed
    vertices = np.concatenate((outside_vertices[::-outside],
                               inside_vertices[::-inside]))

    # Shift the path
    vertices[:, 0] += i * 2.5

    # The codes will be all "LINETO" commands, except for "MOVETO"s at the
    # beginning of each subpath
    all_codes = np.concatenate((codes, codes))

    # Create the Path object
    path = mpath.Path(vertices, all_codes)

    # Add plot it
    patch = mpatches.PathPatch(path, facecolor='#885500', edgecolor='black')
    ax.add_patch(patch)

    ax.annotate("Outside %s, \nInside %s" % (wise(outside), wise(inside)),
                (i * 2.5, -1.5), va="top", ha="center")

ax.set_xlim(-2,10)
ax.set_ylim(-3,2)
ax.set_title('Mmm, donuts!')
ax.set_aspect(1.0)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo to show use of the engineering Formatter.

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import EngFormatter

fig, ax = plt.subplots()
ax.set_xscale('log')
formatter = EngFormatter(unit='Hz', places=1)
ax.xaxis.set_major_formatter(formatter)

xs = np.logspace(1, 9, 100)
ys = (0.8 + 0.4 * np.random.uniform(size=100)) * np.log10(xs)**2
ax.plot(xs, ys)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
75.14 api example code: font_family_rc.py

[source code]

""
You can explicitly set which font family is picked up for a given font
style (e.g., 'serif', 'sans-serif', or 'monospace').

In the example below, we only allow one font family (Tahoma) for the
san-serif font style. You the default family with the font.family rc
param, e.g.,::

   rcParams['font.family'] = 'sans-serif'

and for the font.family you set a list of font styles to try to find
in order::

   rcParams['font.sans-serif'] = ['Tahoma', 'Bitstream Vera Sans', 'Lucida Grande', 'Verdana']

""

# -*- noplot -*-

from matplotlib import
rcParams
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1,2,3], label='test')

ax.legend()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.15 api example code: font_file.py

[source code]

# -*- noplot -*-

""
Although it is usually not a good idea to explicitly point to a single
ttf file for a font instance, you can do so using the
font_manager.FontProperties fname argument (for a more flexible
solution, see the font_family_rc.py and fonts_demo.py examples).

""

import sys
import os
import matplotlib.font_manager as fm
```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1,2,3])

if sys.platform == 'win32':
    fpath = 'C:\Windows\Fonts\Tahoma.ttf'
elif sys.platform.startswith('linux'):
    fonts = ['/usr/share/fonts/truetype/freefont/FreeSansBoldOblique.ttf',
             '/usr/share/fonts/truetype/ttf-liberation/LiberationSans-BoldItalic.ttf',
             '/usr/share/fonts/truetype/msttcorefonts/Comic_Sans_MS.ttf',
            ]
    for fpath in fonts:
        if os.path.exists(fpath):
            break
else:
    fpath = '/Library/Fonts/Tahoma.ttf'

if os.path.exists(fpath):
    prop = fm.FontProperties(fname=fpath)
    fname = os.path.split(fpath)[1]
    ax.set_title('this is a special font: %s' % fname, fontproperties=prop)
else:
    ax.set_title('Demo fails--cannot find a demo font')
ax.set_xlabel('This is the default font')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This example shows how to use a path patch to draw a bunch of rectangles. The technique of using lots of Rectangle instances, or the faster method of using PolyCollections, were implemented before we had proper paths with moveto/lineto, closepoly etc in mpl. Now that we have them, we can draw collections of regularly shaped objects with homogeneous properties more efficiently with a PathCollection. This example makes a histogram -- its more work to set up the vertex arrays at the outset, but it should be much faster for large numbers of objects.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path

fig, ax = plt.subplots()

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 50)
```
```python
# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n

# we need a (numrects x numsides x 2) numpy array for the path helper
# function to build a compound path
XY = np.array([[left, left, right, right], [bottom, top, top, bottom]]).T

# get the Path object
barpath = path.Path.make_compound_path_from_polys(XY)

# make a patch out of it
patch = patches.PathPatch(barpath, facecolor='blue', edgecolor='gray', alpha=0.8)
ax.add_patch(patch)

# update the view limits
ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
Show how to modify the coordinate formatter to report the image "z" value of the nearest pixel given x and y

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

X = 10*np.random.rand(5,3)

t, ax = plt.subplots()
ax.imshow(X, cmap=cm.jet, interpolation='nearest')

numrows, numcols = X.shape

def format_coord(x, y):
    col = int(x+0.5)
    row = int(y+0.5)
    if col>=0 and col<numcols and row>=0 and row<numrows:
        z = X[row,col]
        return 'x=%.4f, y=%.4f, z=%.4f' % (x, y, z)
    else:
        return 'x=%.4f, y=%.4f' % (x, y)
```

"""
```python
# Illustrate the three different join styles

import numpy as np
import matplotlib.pyplot as plt

def plot_angle(ax, x, y, angle, style):
    phi = angle/180*np.pi
    xx = [x+.5,x,x+.5*np.cos(phi)]
    yy = [y,y,y+.5*np.sin(phi)]
    ax.plot(xx, yy, lw=8, color='blue', solid_joinstyle=style)
    ax.plot(xx[1:], yy[1:], lw=1, color='black')
    ax.plot(xx[1::-1], yy[1::-1], lw=1, color='black')
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
ax.plot(xx[1:2], yy[1:2], 'o', color='red', markersize=3)
ax.text(x,y+.2, '%.0f degrees' % angle)

fig, ax = plt.subplots()
ax.set_title('Join style')

for x,style in enumerate((("miter", 'round', 'bevel'))):
    ax.text(x, 5, style)
    for i in range(5):
        plot_angle(ax, x, i, pow(2.0,3+i), style)

ax.set_xlim(-.5,2.75)
ax.set_ylim(-.5,5.5)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.19 api example code: legend_demo.py

import numpy as np
import matplotlib.pyplot as plt
# Make some fake data.
a = b = np.arange(0,3, .02)
c = np.exp(a)
d = c[::-1]

# Create plots with pre-defined labels.
plt.plot(a, c, 'k--', label='Model length')
plt.plot(a, d, 'k:', label='Data length')
plt.plot(a, c+d, 'k', label='Total message length')
legend = plt.legend(loc='upper center', shadow=True, fontsize='x-large')

# Put a nicer background color on the legend.
legend.get_frame().set_facecolor('#00FFCC')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.20  api example code: line_with_text.py
Show how to override basic methods so an artist can contain another artist. In this case, the line contains a Text instance to label it.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines
import matplotlib.transforms as mtransforms
import matplotlib.text as mtext

class MyLine(lines.Line2D):
    def __init__(self, *args, **kwargs):
        # we'll update the position when the line data is set
        self.text = mtext.Text(0, 0, '')
        lines.Line2D.__init__(self, *args, **kwargs)

        # we can't access the label attr until *after* the line is
        # inited
        self.text.set_text(self.get_label())

    def set_figure(self, figure):
        self.text.set_figure(figure)
        lines.Line2D.set_figure(self, figure)

    def set_axes(self, axes):
        self.text.set_axes(axes)
        lines.Line2D.set_axes(self, axes)

    def set_transform(self, transform):
        # 2 pixel offset
        texttrans = transform + mtransforms.Affine2D().translate(2, 2)
        self.text.set_transform(texttrans)
        lines.Line2D.set_transform(self, transform)

    def set_data(self, x, y):
        if len(x):
            self.text.set_position((x[-1], y[-1]))

        lines.Line2D.set_data(self, x, y)

    def draw(self, renderer):
        # draw my label at the end of the line with 2 pixel offset
        lines.Line2D.draw(self, renderer)
        self.text.draw(renderer)

fig, ax = plt.subplots()
```

---

1688 Chapter 75. api Examples
```python
x, y = np.random.rand(2, 20)
line = MyLine(x, y, mfc='red', ms=12, label='line label')
#line.text.set_text('line label')
line.text.set_color('red')
line.text.set_fontsize(16)

ax.add_line(line)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.21 api example code: logo2.py

![Matplotlib Logo](logo2.png)

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cm as cm

mpl.rcParams['xtick.labelsize'] = 10
mpl.rcParams['ytick.labelsize'] = 12
mpl.rcParams['axes.edgecolor'] = 'gray'

axalpha = 0.05
#figcolor = '#EFEFEF'
figcolor = 'white'
dpi = 80
fig = plt.figure(figsize=(6, 1.1), dpi=dpi)
fig.figurePatch.set_edgecolor(figcolor)
fig.figurePatch.set_facecolor(figcolor)

def add_math_background():
    ax = fig.add_axes([0., 0., 1., 1.])
    text = []

```

""
Thanks to Tony Yu <tsyu80@gmail.com> for the logo design
"""
text.append(r"\text{w}^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2} ... \rho_1} - \alpha'_2 U^{1\beta}_{\rho_1 \sigma_2} \}
\frac{\text{d}\rho}{\text{d} t} + \rho \vec{v} \cdot \nabla \vec{v} = -\nabla p + \mu \nabla^2 \vec{v} (0.35, 0.9, 20))
text.append(r"\int_{-\infty}^{\infty} e^{-x^2}dx = \sqrt{\pi}$",
(0.15, 0.3), 25))
text.append(r"\text{E} = mc^2 = \sqrt{{m_0}^2 c^4 + p^2 c^2}$",
(0.7, 0.42), 30))
for eq, (x, y), size in text:
    ax.text(x, y, eq, ha='center', va='center', color="#11557c", alpha=0.25,
    transform=ax.transAxes, fontsize=size)
ax.set_axis_off()
return ax

def add_matplotlib_text(ax):
    ax.text(0.95, 0.5, 'matplotlib', color='#11557c', fontsize=65, 
    ha='right', va='center', alpha=1.0, transform=ax.transAxes)

def add_polar_bar():
    ax = fig.add_axes([0.025, 0.075, 0.2, 0.85], polar=True)
    ax.axesPatch.set_alpha(axalpha)
    ax.set_axisbelow(True)
    N = 7
    arc = 2. * np.pi
    theta = np.arange(0.0, arc, arc/N)
    radii = 10 * np.array([0.2, 0.6, 0.8, 0.7, 0.4, 0.5, 0.8])
    width = np.pi / 4 * np.array([0.4, 0.4, 0.6, 0.8, 0.2, 0.5, 0.3])
    bars = ax.bar(theta, radii, width=width, bottom=0.0)
    for r, bar in zip(radii, bars):
        bar.set_facecolor(cm.jet(r/10.))
        bar.set_alpha(0.6)
    for label in ax.get_xticklabels() + ax.get_yticklabels():
        label.set_visible(False)
    for line in ax.get_ygridlines() + ax.get_xgridlines():
        line.set_lw(0.8)
        line.set_alpha(0.9)
        line.set_ls('-')
        line.set_color('0.5')
    ax.set_ylim(np.arange(1, 9, 2))
    ax.set_rmax(9)

if __name__ == '__main__':
    main_axes = add_math_background()
    add_polar_bar()
    add_matplotlib_text(main_axes)
    plt.show()
Load a mathtext image as numpy array

```
import numpy as np
import matplotlib.mathtext as mathtext
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rc('image', origin='upper')

parser = mathtext.MathTextParser("Bitmap")

parser.to_png('test2.png', r'\left[\left\lfloor\frac{5}{\frac{\left(3\right)}{4}} y\right)\right]\right', color='green', fontsize=14, dpi=100)

rgba1, depth1 = parser.to_rgba(r'IQ: $\sigma_i=15$', color='blue', fontsize=20, dpi=200)
rgba2, depth2 = parser.to_rgba(r'some other string', color='red', fontsize=20, dpi=200)
```
fig = plt.figure()
fig.figimage(rgba1.astype(float)/255., 100, 100)
fig.figimage(rgba2.astype(float)/255., 100, 300)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.23 api example code: patch_collection.py

import numpy as np
import matplotlib
from matplotlib.patches import Circle, Wedge, Polygon
from matplotlib.collections import PatchCollection
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

resolution = 50  # the number of vertices
N = 3
x = np.random.rand(N)

...
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
patches = []
for x1, y1, r in zip(x, y, radii):
    circle = Circle((x1, y1), r)
    patches.append(circle)

x = np.random.rand(N)
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
theta1 = 360.0*np.random.rand(N)
theta2 = 360.0*np.random.rand(N)
for x1, y1, r, t1, t2 in zip(x, y, radii, theta1, theta2):
    wedge = Wedge((x1, y1), r, t1, t2)
    patches.append(wedge)

    # Some limiting conditions on Wedge
patches += [
    Wedge((.3, .7), .1, 0, 360),  # Full circle
    Wedge((.7, .8), .2, 0, 360, width=0.05),  # Full ring
    Wedge((.8, .3), .2, 0, 45),  # Full sector
    Wedge((.8, .3), .2, 45, 90, width=0.10),  # Ring sector
]

for i in range(N):
    polygon = Polygon(np.random.rand(N, 2), True)
    patches.append(polygon)

colors = 100*np.random.rand(len(patches))
p = PatchCollection(patches, cmap=matplotlib.cm.jet, alpha=0.4)
p.set_array(np.array(colors))
ax.add_collection(p)
plt.colorbar(p)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
75.24 api example code: power_norm_demo.py

```python
#!/usr/bin/python
from matplotlib import pyplot as plt
import matplotlib.colors as mcolors
import numpy as np
from numpy.random import multivariate_normal

data = np.vstack([multivariate_normal([10, 10], [[3, 5], [4, 2]], size=100000),
                  multivariate_normal([30, 20], [[2, 3], [1, 3]], size=1000)])

gammas = [0.8, 0.5, 0.3]
xgrid = np.floor((len(gammas) + 1.) / 2)
ygrid = np.ceil((len(gammas) + 1.) / 2)

plt.subplot(xgrid, ygrid, 1)
plt.title('Linear normalization')
plt.hist2d(data[:,0], data[:,1], bins=100)

for i, gamma in enumerate(gammas):
    plt.subplot(xgrid, ygrid, i + 2)
    plt.title('Power law normalization\n($\gamma=%.1f$)' % gamma)
```
```python
plt.hist2d(data[:, 0], data[:, 1],
           bins=100, norm=mcolors.PowerNorm(gamma))
plt.subplots_adjust(hspace=0.39)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.25 api example code: quad_bezier.py

```
ax.add_patch(pp1)
ax.plot([0.75], [0.25], "ro")
ax.set_title('The red point should be on the path')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.26 api example code: radar_chart.py

5-Factor Solution Profiles Across Four Scenarios
Example of creating a radar chart (a.k.a. a spider or star chart) [1]_.

Although this example allows a frame of either 'circle' or 'polygon', polygon frames don't have proper gridlines (the lines are circles instead of polygons). It's possible to get a polygon grid by setting GRIDLINE_INTERPOLATION_STEPS in matplotlib.axis to the desired number of vertices, but the orientation of the polygon is not aligned with the radial axes.


```
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.path import Path
from matplotlib.spines import Spine
from matplotlib.projections.polar import PolarAxes
from matplotlib.projections import register_projection

def radar_factory(num_vars, frame='circle'):
    """Create a radar chart with `num_vars` axes.
    This function creates a RadarAxes projection and registers it.
    Parameters
    ----------
    num_vars : int
        Number of variables for radar chart.
    frame : {'circle' | 'polygon'}
        Shape of frame surrounding axes.
    ""
    # calculate evenly-spaced axis angles
    theta = 2*np.pi * np.linspace(0, 1-1./num_vars, num_vars)
    # rotate theta such that the first axis is at the top
    theta += np.pi/2

def draw_poly_patch(self):
    verts = unit_poly_verts(theta)
    return plt.Polygon(verts, closed=True, edgecolor='k')

def draw_circle_patch(self):
    # unit circle centered on (0.5, 0.5)
    return plt.Circle((0.5, 0.5), 0.5)

patch_dict = {'polygon': draw_poly_patch, 'circle': draw_circle_patch}
if frame not in patch_dict:
    raise ValueError('unknown value for `frame`: %s' % frame)

class RadarAxes(PolarAxes):
    name = 'radar'
```
# use 1 line segment to connect specified points
RESOLUTION = 1
# define draw_frame method
draw_patch = patch_dict[frame]

def fill(self, *args, **kwargs):
    """Override fill so that line is closed by default""
    closed = kwargs.pop('closed', True)
    return super(RadarAxes, self).fill(closed=closed, *args, **kwargs)

def plot(self, *args, **kwargs):
    """Override plot so that line is closed by default""
    lines = super(RadarAxes, self).plot(*args, **kwargs)
    for line in lines:
        self._close_line(line)

def _close_line(self, line):
    x, y = line.get_data()
    # FIXME: markers at x[0], y[0] get doubled-up
    if x[0] != x[-1]:
        x = np.concatenate((x, [x[0]]))
        y = np.concatenate((y, [y[0]]))
        line.set_data(x, y)

def set_varlabels(self, labels):
    self.set_thetagrids(theta * 180/np.pi, labels)

def _gen_axes_patch(self):
    return self.draw_patch()

def _gen_axes_spines(self):
    if frame == 'circle':
        return PolarAxes._gen_axes_spines(self)
    # The following is a hack to get the spines (i.e. the axes frame)
    # to draw correctly for a polygon frame.
    # spine_type must be 'left', 'right', 'top', 'bottom', or 'circle'.
    spine_type = 'circle'
    verts = unit_poly_verts(theta)
    # close off polygon by repeating first vertex
    verts.append(verts[0])
    path = Path(verts)
    spine = Spine(self, spine_type, path)
    spine.set_transform(self.transAxes)
    return {'polar': spine}

register_projection(RadarAxes)
return theta

def unit_poly_verts(theta):
    """Return vertices of polygon for subplot axes.

This polygon is circumscribed by a unit circle centered at (0.5, 0.5)

```python
x0, y0, r = [0.5] * 3
verts = [(r*np.cos(t) + x0, r*np.sin(t) + y0) for t in theta]
return verts
```

def example_data():
    # The following data is from the Denver Aerosol Sources and Health study.
    #
    # The data are pollution source profile estimates for five modeled pollution
    # sources (e.g., cars, wood-burning, etc) that emit 7-9 chemical species.
    # The radar charts are experimented with here to see if we can nicely
    # visualize how the modeled source profiles change across four scenarios:
    # 1) No gas-phase species present, just seven particulate counts on
    # Sulfate
    # Nitrate
    # Elemental Carbon (EC)
    # Organic Carbon fraction 1 (OC)
    # Organic Carbon fraction 2 (OC2)
    # Organic Carbon fraction 3 (OC3)
    # Pyrolized Organic Carbon (OP)
    # 2) Inclusion of gas-phase specie carbon monoxide (CO)
    # 3) Inclusion of gas-phase specie ozone (O3).
    # 4) Inclusion of both gas-phase species is present...
    data = {
        'column names':
            ['Sulfate', 'Nitrate', 'EC', 'OC1', 'OC2', 'OC3', 'OP', 'CO', 'O3'],
        'Basecase':
            [[0.88, 0.01, 0.03, 0.03, 0.00, 0.06, 0.01, 0.00, 0.00],
             [0.07, 0.95, 0.04, 0.05, 0.00, 0.02, 0.01, 0.00, 0.00],
             [0.01, 0.02, 0.85, 0.19, 0.05, 0.10, 0.90, 0.00, 0.00],
             [0.02, 0.01, 0.07, 0.01, 0.21, 0.12, 0.98, 0.00, 0.00],
             [0.01, 0.01, 0.02, 0.71, 0.74, 0.70, 0.00, 0.00, 0.00]],
        'With CO':
            [[0.88, 0.02, 0.02, 0.02, 0.00, 0.05, 0.00, 0.05, 0.00],
             [0.08, 0.94, 0.04, 0.02, 0.00, 0.01, 0.12, 0.04, 0.00],
             [0.01, 0.01, 0.79, 0.10, 0.00, 0.05, 0.00, 0.31, 0.00],
             [0.00, 0.02, 0.03, 0.38, 0.31, 0.31, 0.00, 0.59, 0.00],
             [0.02, 0.02, 0.11, 0.47, 0.69, 0.58, 0.88, 0.00, 0.00]],
        'With O3':
            [[0.89, 0.01, 0.07, 0.00, 0.00, 0.05, 0.00, 0.00, 0.03],
             [0.07, 0.95, 0.05, 0.04, 0.00, 0.02, 0.12, 0.00, 0.00],
             [0.01, 0.02, 0.86, 0.27, 0.16, 0.19, 0.00, 0.00, 0.00],
             [0.01, 0.03, 0.00, 0.32, 0.29, 0.27, 0.00, 0.00, 0.95],
             [0.02, 0.00, 0.03, 0.37, 0.56, 0.47, 0.87, 0.00, 0.00]],
        'CO & O3':
            [[0.87, 0.01, 0.08, 0.00, 0.00, 0.04, 0.00, 0.00, 0.01],
             [0.09, 0.95, 0.02, 0.03, 0.00, 0.01, 0.13, 0.06, 0.00],
             [0.01, 0.02, 0.71, 0.24, 0.13, 0.16, 0.00, 0.50, 0.00],
             [0.01, 0.03, 0.00, 0.28, 0.24, 0.23, 0.00, 0.44, 0.88],
```
```python
if __name__ == '__main__':
    N = 9
    theta = radar_factory(N, frame='polygon')

    data = example_data()
    spoke_labels = data.pop('column names')

    fig = plt.figure(figsize=(9, 9))
    fig.subplots_adjust(wspace=0.25, hspace=0.20, top=0.85, bottom=0.05)

    colors = ['b', 'r', 'g', 'm', 'y']
    # Plot the four cases from the example data on separate axes
    for n, title in enumerate(data.keys()):
        ax = fig.add_subplot(2, 2, n+1, projection='radar')
        plt.rgrids([0.2, 0.4, 0.6, 0.8])
        ax.set_title(title, weight='bold', size='medium', position=(0.5, 1.1),
                      horizontalalignment='center', verticalalignment='center')
        for d, color in zip(data[title], colors):
            ax.plot(theta, d, color=color)
            ax.fill(theta, d, facecolor=color, alpha=0.25)
            ax.set_varlabels(spoke_labels)

    # add legend relative to top-left plot
    plt.subplot(2, 2, 1)
    labels = ('Factor 1', 'Factor 2', 'Factor 3', 'Factor 4', 'Factor 5')
    legend = plt.legend(labels, loc=(0.9, .95), labelspacing=0.1)
    plt.setp(legend.get_texts(), fontsize='small')

    plt.figtext(0.5, 0.965, '5-Factor Solution Profiles Across Four Scenarios',
                 ha='center', color='black', weight='bold', size='large')

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
The default settings produce a diagram like this.
Flow Diagram of a Widget

Widget A

25%
0% First 10% Second 20%
60%
Third 5%
Fourth 15%
Fifth 10%
Hurray! 40%
"""Demonstrate the Sankey class by producing three basic diagrams."
"""
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.sankey import Sankey

# Example 1 -- Mostly defaults
# This demonstrates how to create a simple diagram by implicitly calling the
# Sankey.add() method and by appending finish() to the call to the class.
Sankey(flows=[0.25, 0.15, 0.60, -0.20, -0.15, -0.05, -0.50, -0.10],
       labels=['', '', '', 'First', 'Second', 'Third', 'Fourth', 'Fifth'],
       orientations=[-1, 1, 0, 1, 1, 0, -1]).finish()
plt.title("The default settings produce a diagram like this.")

# Notice:
# 1. Axes weren't provided when Sankey() was instantiated, so they were
#     created automatically.
# 2. The scale argument wasn't necessary since the data was already
#     normalized.
# 3. By default, the lengths of the paths are justified.

# Example 2
# This demonstrates:
# 1. Setting one path longer than the others
# 2. Placing a label in the middle of the diagram
# 3. Using the the scale argument to normalize the flows
# 4. Implicitly passing keyword arguments to PathPatch()
# 5. Changing the angle of the arrow heads
# 6. Changing the offset between the tips of the paths and their labels
# 7. Formatting the numbers in the path labels and the associated unit
# 8. Changing the appearance of the patch and the labels after the figure is
# created

```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
        title="Flow Diagram of a Widget")
sankey = Sankey(ax=ax, scale=0.01, offset=0.2, head_angle=180,
        format='%.0f', unit='%')
sankey.add(flows=[25, 0, 60, -10, -20, -5, -15, -10, -40],
        labels = ['First', 'Second', 'Third', 'Fourth',
               'Fifth', 'Hurray!'],
        orientations=[-1, 1, 0, 1, 1, -1, -1, 0],
        pathlengths = [0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25,
                       0.25],
        patchlabel="Widget\n",
        alpha=0.2, lw=2.0)  # Arguments to matplotlib.patches.PathPatch()

diagrams = sankey.finish()
diagrams[0].patch.set_facecolor('#37c959')
diagrams[0].texts[-1].set_color(r')
diagrams[0].text.set_fontweight('bold')

# Notice:
# 1. Since the sum of the flows is nonzero, the width of the trunk isn't
#    uniform. If verbose.level is helpful (in matplotlibrc), a message is
#    given in the terminal window.
# 2. The second flow doesn't appear because its value is zero. Again, if
#    verbose.level is helpful, a message is given in the terminal window.

# Example 3
# This demonstrates:
# 1. Connecting two systems
# 2. Turning off the labels of the quantities
# 3. Adding a legend
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
        title="Two Systems")
flows = [0.25, 0.15, 0.60, -0.10, -0.05, -0.25, -0.15, -0.10, -0.35]
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=flows, label='one',
            orientations=[-1, 1, 0, 1, 1, -1, -1, 0])
sankey.add(flows=[-0.25, 0.15, 0.1], fc='#37c959', label='two',
            orientations=[-1, -1, -1], prior=0, connect=(0, 0))

diagrams = sankey.finish()
diagrams[-1].patch.set_hatch('/')
plt.legend(loc='best')

# Notice that only one connection is specified, but the systems form a
# circuit since: (1) the lengths of the paths are justified and (2) the
# orientation and ordering of the flows is mirrored.

plt.show()
```
Why would you want to do this?
(But you could.)

```python
"""Demonstrate/test the Sankey class by producing a long chain of connections.
"""
from itertools import cycle
import matplotlib.pyplot as plt
from matplotlib.sankey import Sankey

links_per_side = 6

def side(sankey, n=1):
    """Generate a side chain."""
    prior = len(sankey.diagrams)
    colors = cycle(['orange', 'b', 'g', 'r', 'c', 'm', 'y'])
    for i in range(0, 2*n, 2):
        sankey.add(flows=[1, -1], orientations=[-1, -1],
                   patchlabel=str(prior+i), facecolor=next(colors),
                   prior=prior+i-1, connect=(1, 0), alpha=0.5)
```

75.28 api example code: sankey_demo_links.py
```python
def corner(sankey):
    """Generate a corner link."""
    prior = len(sankey.diagrams)
    sankey.add(flows=[1, -1], orientations=[0, 1],
               patchlabel=str(prior), facecolor='k',
               prior=prior-1, connect=(1, 0), alpha=0.5)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                      title="Why would you want to do this?
                      (But you could.)")
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=[1, -1], orientations=[0, 1],
           patchlabel="0", facecolor='k',
           rotation=45)
side(sankey, n=links_per_side)
side(sankey, n=links_per_side)
side(sankey, n=links_per_side)
side(sankey, n=links_per_side)
side(sankey, n=links_per_side)
sankey.finish()
    # Notice:
    # 1. The alignment doesn't drift significantly (if at all; with 16007
    #     subdiagrams there is still closure).
    # 2. The first diagram is rotated 45 deg, so all other diagrams are rotated
    #     accordingly.

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

from __future__ import print_function

__author__ = "Yannick Copin <ycopin@ipnl.in2p3.fr>"
__version__ = "Time-stamp: <10/02/2010 16:49 ycopin@lyopc548.in2p3.fr>"

import numpy as np

def sankey(ax,
    outputs=[100.], outlabels=None,
    inputs=[100.], inlabels=''
    dx=40, dy=10, outangle=45, w=3, inangle=30, offset=2, **kwargs):
    """Draw a Sankey diagram.

    outputs: array of outputs, should sum up to 100%
    outlabels: output labels (same length as outputs),
    or None (use default labels) or """ (no labels)
    inputs and inlabels: similar for inputs
dx: horizontal elongation
dy: vertical elongation"""
outangle: output arrow angle [deg]
w: output arrow shoulder
inangle: input dip angle
offset: text offset
**kwargs: propagated to Patch (e.g., fill=False)

Return (patch,[intexts,outtexts]).

```python
import matplotlib.patches as mpatches
from matplotlib.path import Path
outs = np.absolute(outputs)
outsigns = np.sign(outputs)
outsigns[-1] = 0  # Last output

ins = np.absolute(inputs)\ninsigns = np.sign(inputs)
insigns[0] = 0  # First input

assert sum(outs) == 100, "Outputs don't sum up to 100%"
assert sum(ins) == 100, "Inputs don't sum up to 100%"

def add_output(path, loss, sign=1):
    h = (loss/2 + w)*np.tan(outangle/180. * np.pi)  # Arrow tip height
    move, (x, y) = path[-1]  # Use last point as reference
    if sign == 0:
        # Final loss (horizontal)
        path.extend(((Path.LINETO, [x+dx, y]),
                      (Path.LINETO, [x+dx, y+gain/2]),
                      (Path.LINETO, [x+dx*h, y+gain]),
                      (Path.LINETO, [x+dx, y+gain])))
        outtips.append((sign, path[-3][1]))
    else:
        # Intermediate loss (vertical)
        path.extend(((Path.CURVE4, [x+dx/2, y]),
                      (Path.CURVE4, [x+dx, y]),
                      (Path.CURVE4, [x+dx, y+sign*dy]),
                      (Path.LINETO, [x+dx-w, y+sign*dy]),
                      (Path.LINETO, [x+dx+loss/2, y+sign*(dy+h)]),  # Tip
                      (Path.LINETO, [x+dx+loss+w, y+sign*dy]),
                      (Path.LINETO, [x+dx+loss, y+sign*dy]),
                      (Path.CURVE3, [x+dx+loss, y+sign*loss]),
                      (Path.CURVE3, [x+dx/2+loss, y+sign*loss])))
        outtips.append((sign, path[-5][1]))

def add_input(path, gain, sign=1):
    h = (gain/2)*np.tan(inangle/180. * np.pi)  # Dip depth
    move, (x, y) = path[-1]  # Use last point as reference
    if sign == 0:
        # First gain (horizontal)
        path.extend(((Path.LINETO, [x-dx, y]),
                      (Path.LINETO, [x-dx+h, y+gain]),
                      (Path.LINETO, [x-dx, y+gain/2]),
                      (Path.LINETO, [x-dx*h, y+gain]),
                      (Path.LINETO, [x-dx, y+gain])))
        xd, yd = path[-2][1]  # Dip position
        indips.append((sign, [xd-h, yd]))
```
```python
else:  # Intermediate gain (vertical)
    path.extend([(Path.CURVE4, [x-dx/2, y]),
                  (Path.CURVE4, [x-dx, y]),
                  (Path.CURVE4, [x-dx, y+sign*dy]),
                  (Path.LINETO, [x-dx-gain/2, y+sign*(dy-h)]),  # Dip
                  (Path.LINETO, [x-dx-gain, y+sign*dy]),
                  (Path.CURVE3, [x-dx-gain, y-sign*gain]),
                  (Path.CURVE3, [x-dx/2-gain, y-sign*gain])])
    xd, yd = path[-4][1]  # Dip position
    indips.append((sign, [xd, yd+sign*h]))

outtips = []  # Output arrow tip dir. and positions
urpath = [(Path.MOVETO, [0, 100])]  # 1st point of upper right path
lrpath = [(Path.LINETO, [0, 0])]  # 1st point of lower right path
for loss, sign in zip(outs, outsins):
    add_output(sign>=0 and urpath or lrpath, loss, sign=sign)

indips = []  # Input arrow tip dir. and positions
llpath = [(Path.LINETO, [0, 0])]  # 1st point of lower left path
ulpath = [(Path.MOVETO, [0, 100])]  # 1st point of upper left path
for gain, sign in reversed(list(zip(ins, insigns))):
    add_input(sign<=0 and llpath or ulpath, gain, sign=sign)

def revert(path):
    """A path is not just revertable by path[::-1] because of Bezier curves."""
    rpath = []
    nextmove = Path.LINETO
    for move, pos in path[::-1]:
        rpath.append((nextmove, pos))
        nextmove = move
    return rpath

# Concatenate subpathes in correct order
path = urpath + revert(lrpath) + llpath + revert(ulpath)

codes, verts = zip(*path)
verts = np.array(verts)

# Path patch
path = Path(verts, codes)
patch = mpatches.PathPatch(path, **kwargs)
ax.add_patch(patch)

if False:  # DEBUG
    print("urpath", urpath)
    print("lrpath", revert(lrpath))
    print("llpath", llpath)
    print("ulpath", revert(ulpath))
    xs, ys = zip(*verts)
    ax.plot(xs, ys, 'go-')
```

# Labels
def set_labels(labels, values):
    """Set or check labels according to values."""
    if labels == '':  # No labels
        return labels
    elif labels is None:  # Default labels
        return ['%2d%%' % val for val in values]
    else:
        assert len(labels) == len(values)
        return labels

def put_labels(labels, positions, output=True):
    """Put labels to positions."""
    texts = []
    lbls = output and labels or labels[::-1]
    for i, label in enumerate(lbls):
        s, (x, y) = positions[i]  # Label direction and position
        if s == 0:
            t = ax.text(x, y, label, ha=output and 'left' or 'right', va='center')
        elif s > 0:
            t = ax.text(x, y, label, ha='center', va='bottom')
        else:
            t = ax.text(x, y, label, ha='center', va='top')
        texts.append(t)
    return texts

outlabels = set_labels(outlabels, outs)
outtexts = put_labels(outlabels, outtips, output=True)

inlabels = set_labels(inlabels, ins)
intexts = put_labels(inlabels, indips, output=False)

# Axes management
ax.set_xlim(verts[:, 0].min()-dx, verts[:, 0].max()+dx)
ax.set_ylim(verts[:, 1].min()-dy, verts[:, 1].max()+dy)
ax.set_aspect('equal', adjustable='datalim')

return patch, [intexts, outtexts]

if __name__ == '__main__':
    import matplotlib.pyplot as plt

outputs = [10., -20., 5., 15., -10., 40.]
outlabels = ['First', 'Second', 'Third', 'Fourth', 'Fifth', 'Hurray!']
outlabels = [s+'\n\n%d%%' % abs(l) for l, s in zip(outputs, outlabels)]

inputs = [60., -25., 15.]

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[], title="Sankey diagram")
patch, (intexts, outtexts) = sankey(ax, outputs=outputs,
    outlabels=outlabels, inputs=inputs,
    inlabels=None, fc='g', alpha=0.2)
outtexts[1].set_color('r')
outtexts[-1].set_fontweight('bold')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demonstrate the Sankey class with a practical example of a Rankine power cycle.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.sankey import Sankey
```
fig = plt.figure(figsize=(8, 9))
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
title="Rankine Power Cycle: Example 8.6 from Moran and Shapiro
Hdot = [260.431, 35.078, 180.794, 221.115, 22.700,
142.361, 10.193, 10.210, 43.670, 44.312,
68.631, 10.758, 10.758, 0.017, 0.642,
232.121, 44.559, 100.613, 132.168]  # MW
sankey = Sankey(ax=ax, format='%.3G', unit='MW', gap=0.5, scale=1.0/Hdot[0])
sankey.add(patchlabel='

Pump 1', rotation=90, facecolor='#37c959',
flows=[Hdot[13], Hdot[6], -Hdot[7]],
labels=['Shaft power', '', None],
pathlengths=[0.4, 0.883, 0.25],
orientations=[1, -1, 0])
sankey.add(patchlabel='

Open heater', facecolor='#37c959',
flows=[Hdot[11], Hdot[7], Hdot[4], -Hdot[8]],
labels=[None, '', None, None],
pathlengths=[0.25, 0.25, 1.93, 0.25],
orientations=[1, 0, -1, 0], prior=0, connect=(2, 1))
sankey.add(patchlabel='

Pump 2', facecolor='#37c959',
flows=[Hdot[14], Hdot[8], -Hdot[9]],
labels=['Shaft power', '', None],
pathlengths=[0.4, 0.25, 0.25],
orientations=[1, 0, 0], prior=1, connect=(3, 1))
sankey.add(patchlabel='Closed heater', trunklength=2.914, fc='#37c959',
flows=[Hdot[9], Hdot[11], -Hdot[10]],
labels=['', '', None, None],
pathlengths=[0.25, 1.543, 0.25, 0.25],
orientations=[0, -1, 1, -1], prior=2, connect=(2, 0))
sankey.add(patchlabel='Trap', facecolor='#37c959', trunklength=5.102,
flows=[Hdot[11], -Hdot[12]],
labels=['\n', None],
pathlengths=[1.0, 1.01],
orientations=[1, 1], prior=3, connect=(2, 0))
sankey.add(patchlabel='Steam\ngenerator', facecolor='#ff5555',
flows=[Hdot[15], -Hdot[10], Hdot[2], -Hdot[3], -Hdot[0]],
labels=['Heat rate', '', '', None, None],
pathlengths=0.25,
orientations=[1, 0, -1, -1, -1], prior=3, connect=(3, 1))
sankey.add(patchlabel='Turbine 1', facecolor='#37c959',
flows=[Hdot[0], -Hdot[16], -Hdot[1], -Hdot[2]],
labels=['', None, None, None],
pathlengths=[0.25, 0.153, 1.543, 0.25],
orientations=[0, 1, -1, -1], prior=5, connect=(4, 0))
sankey.add(patchlabel='Reheat', facecolor='#37c959',
flows=[Hdot[2], -Hdot[17]],
labels=[None, None],
pathlengths=[0.725, 0.25],
orientations=[-1, 0], prior=6, connect=(3, 0))
sankey.add(patchlabel='Turbine 2', trunklength=3.212, facecolor='#37c959',
flows=[Hdot[3], Hdot[16], -Hdot[5], -Hdot[4], -Hdot[17]],
labels=[None, 'Shaft power', None, '', 'Shaft power'],
pathlengths=[0.751, 0.15, 0.25, 1.93, 0.25],
75.30. api example code: sankey_demo_rankine.py
orientations=[0, -1, 0, -1, 1], prior=6, connect=(1, 1))
sankey.add(patchlabel='Condenser', facecolor='#58b1fa', trunklength=1.764,
flows=[Hdot[5], -Hdot[18], -Hdot[6]],
labels=['', 'Heat rate', None],
pathlengths=[0.45, 0.25, 0.883],
orientations=[-1, 1, 0], prior=8, connect=(2, 0))
diagrams = sankey.finish()
for diagram in diagrams:
    diagram.text.set_fontweight('bold')
    diagram.text.set_fontsize('10')
    for text in diagram.texts:
        text.set_fontsize('10')
# Notice that the explicit connections are handled automatically, but the
# implicit ones currently are not. The lengths of the paths and the trunks
# must be adjusted manually, and that is a bit tricky.

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.31 api example code: scatter_piecharts.py
This example makes custom 'pie charts' as the markers for a scatter plot.

Thanks to Manuel Metz for the example.

```python
import math
import numpy as np
import matplotlib.pyplot as plt

# first define the ratios
r1 = 0.2  # 20%
r2 = r1 + 0.4  # 40%

# define some sizes of the scatter marker
sizes = [60, 80, 120]

# calculate the points of the first pie marker
# these are just the origin (0,0) +
# some points on a circle cos,sin
x = [0] + np.cos(np.linspace(0, 2*math.pi*r1, 10)).tolist()
y = [0] + np.sin(np.linspace(0, 2*math.pi*r1, 10)).tolist()

xy1 = list(zip(x, y))
s1 = max(max(x), max(y))

# ...
x = [0] + np.cos(np.linspace(2*math.pi*r1, 2*math.pi*r2, 10)).tolist()
y = [0] + np.sin(np.linspace(2*math.pi*r1, 2*math.pi*r2, 10)).tolist()

xy2 = list(zip(x, y))
s2 = max(max(x), max(y))

x = [0] + np.cos(np.linspace(2*math.pi*r2, 2*math.pi, 10)).tolist()
y = [0] + np.sin(np.linspace(2*math.pi*r2, 2*math.pi, 10)).tolist()

xy3 = list(zip(x, y))
s3 = max(max(x), max(y))

fig, ax = plt.subplots()
ax.scatter(np.arange(3), np.arange(3), marker=(xy1, 0),
           s=[s1*s1* for _ in sizes], facecolor='blue')
ax.scatter(np.arange(3), np.arange(3), marker=(xy2, 0),
           s=[s2*s2* for _ in sizes], facecolor='green')
ax.scatter(np.arange(3), np.arange(3), marker=(xy3, 0),
           s=[s3*s3* for _ in sizes], facecolor='red')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
This serves as an intensive exercise of matplotlib's transforms and custom projection API. This example produces a so-called SkewT-logP diagram, which is a common plot in meteorology for displaying vertical profiles of temperature. As far as matplotlib is concerned, the complexity comes from having X and Y axes that are not orthogonal. This is handled by including a skew component to the basic Axes transforms. Additional complexity comes in handling the fact that the upper and lower X-axes have different data ranges, which necessitates a bunch of custom classes for ticks, spines, and the axis to handle this.

```python
from matplotlib.axes import Axes
import matplotlib.transforms as transforms
```
import matplotlib.axis as maxis
import matplotlib.spines as mspines
import matplotlib.path as mpath
from matplotlib.projections import register_projection

# The sole purpose of this class is to look at the upper, lower, or total
# interval as appropriate and see what parts of the tick to draw, if any.
class SkewXTick(maxis.XTick):
    def draw(self, renderer):
        if not self.get_visible():
            return
        renderer.open_group(self.__name__)
        lower_interval = self.axes.xaxis.lower_interval
        upper_interval = self.axes.xaxis.upper_interval

        if self.gridOn and transforms.interval_contains(
            self.axes.xaxis.get_view_interval(), self.get_loc()):
            self.gridline.draw(renderer)

        if transforms.interval_contains(lower_interval, self.get_loc()):
            if self.tick1On:
                self.tick1line.draw(renderer)
            if self.label1On:
                self.label1.draw(renderer)

        if transforms.interval_contains(upper_interval, self.get_loc()):
            if self.tick2On:
                self.tick2line.draw(renderer)
            if self.label2On:
                self.label2.draw(renderer)

        renderer.close_group(self.__name__)

# This class exists to provide two separate sets of intervals to the tick,
# as well as create instances of the custom tick
class SkewXAxis(maxis.XAxis):
    def __init__(self, *args, **kwargs):
        maxis.XAxis.__init__(self, *args, **kwargs)
        self.upper_interval = 0.0, 1.0

    def _get_tick(self, major):
        return SkewXTick(self.axes, 0, '', major=major)

    @property
    def lower_interval(self):
        return self.axes.viewLim.intervalx

    def get_view_interval(self):
        return self.upper_interval[0], self.axes.viewLim.intervalx[1]

# This class exists to calculate the separate data range of the
# upper X-axis and draw the spine there. It also provides this range
# to the X-axis artist for ticking and gridlines

class SkewSpine(mspines.Spine):
    def _adjust_location(self):
        trans = self.axes.transDataToAxes.inverted()
        if self.spine_type == 'top':
            yloc = 1.0
        else:
            yloc = 0.0
        left = trans.transform_point((0.0, yloc))[0]
        right = trans.transform_point((1.0, yloc))[0]

        pts = self._path.vertices
        pts[0, 0] = left
        pts[1, 0] = right
        self.axis.upper_interval = (left, right)

# This class handles registration of the skew-xaxes as a projection as well
# as setting up the appropriate transformations. It also overrides standard
# spines and axes instances as appropriate.

class SkewXAxes(Axes):
    # The projection must specify a name. This will be used be the
    # user to select the projection, i.e. ``subplot(111,
    # projection='skewx')``.
    name = 'skewx'

    def _init_axis(self):
        #Taken from Axes and modified to use our modified X-axis
        self.xaxis = SkewXAxis(self)
        self.spines['top'].register_axis(self.xaxis)
        self.spines['bottom'].register_axis(self.xaxis)
        self.yaxis = maxis.YAxis(self)
        self.spines['left'].register_axis(self.yaxis)
        self.spines['right'].register_axis(self.yaxis)

    def _gen_axes_spines(self):
        spines = {'top':SkewSpine.linear_spine(self, 'top'),
                  'bottom':mspines.Spine.linear_spine(self, 'bottom'),
                  'left':mspines.Spine.linear_spine(self, 'left'),
                  'right':mspines.Spine.linear_spine(self, 'right')}
        return spines

    def _set_lim_and_transforms(self):
        ""
        This is called once when the plot is created to set up all the
        transforms for the data, text and grids.
        ""
        rot = 30

        #Get the standard transform setup from the Axes base class
        Axes._set_lim_and_transforms(self)
# Need to put the skew in the middle, after the scale and limits,  
# but before the transAxes. This way, the skew is done in Axes  
# coordinates thus performing the transform around the proper origin  
# We keep the pre-transAxes transform around for other users, like the  
# spines for finding bounds  
self.transDataToAxes = self.transScale + (self.transLimits +  
transfforms.Affine2D().skew_deg(rot, 0))  

# Create the full transform from Data to Pixels  
self.transData = self.transDataToAxes + self.transAxes  

# Blended transforms like this need to have the skewing applied using  
# both axes, in axes coords like before.  
self._xaxis_transform = (transfforms.blended_transform_factory(  
    self.transScale + self.transLimits,  
    transfforms.IdentityTransform()) +  
    transfforms.Affine2D().skew_deg(rot, 0)) + self.transAxes  

# Now register the projection with matplotlib so the user can select  
# it.  
register_projection(SkewXAxes)  

if __name__ == '__main__':  
    # Now make a simple example using the custom projection.  
    from matplotlib.ticker import ScalarFormatter, MultipleLocator  
    from matplotlib.collections import LineCollection  
    import matplotlib.pyplot as plt  
    from StringIO import StringIO  
    import numpy as np  

    # Some examples data  
    data_txt = '978.0 345 7.8 0.8 61 4.16 325 14 282.7 294.6 283.4
971.0 404 7.2 0.2 61 4.01 327 17 282.7 294.2 283.4
946.7 610 5.2 -1.8 61 3.56 335 26 282.8 293.0 283.4
944.0 634 5.0 -2.0 61 3.51 336 27 282.8 292.9 283.4
925.0 798 3.4 -2.6 65 3.56 340 32 282.8 292.7 283.4
911.8 914 2.4 -2.7 69 3.56 345 37 282.9 292.9 283.5
906.0 966 2.0 -2.7 71 3.47 348 39 283.0 293.0 283.6
877.9 1219 0.4 -3.2 77 3.46 335 26 283.9 293.9 284.5
850.0 1478 -1.3 -3.7 74 3.44 340 27 284.8 294.8 285.4
841.0 1563 -1.9 -3.8 78 3.45 358 45 285.0 295.0 285.6
823.0 1736 1.4 -0.7 86 4.44 353 42 290.3 303.3 291.0
813.6 1829 4.5 1.2 80 5.17 350 40 294.5 309.8 295.4
809.0 1875 6.0 2.2 77 5.57 347 39 296.6 313.2 297.6
798.0 1988 7.4 -0.6 57 4.61 340 35 299.2 313.3 300.1
791.0 2061 7.6 -1.4 53 4.39 335 33 300.2 313.6 301.0
783.9 2134 7.0 -1.7 54 4.32 330 31 300.4 313.6 301.2
755.1 2438 4.8 -3.1 57 4.06 300 24 301.2 313.7 301.9
727.3 2743 2.5 -4.4 60 3.81 285 29 301.9 313.8 302.6
700.5 3048 0.2 -5.8 64 3.57 275 31 302.7 313.8 303.3
700.0 3054 0.2 -5.8 64 3.56 280 31 302.7 313.8 303.3
698.0 3077 0.0 -6.0 64 3.52 280 31 302.7 313.7 303.4

75.32. api example code: skewt.py 1719
| x  | y       | a | b   | c | d | e | f | g | h   | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| 687.0 | 3204 | -0.1 | -7.1 | 59 | 3.28 | 281 | 31 | 304.0 | 314.3 | 304.6 |
| 648.9 | 3658 | -3.2 | -10.9 | 55 | 2.59 | 285 | 30 | 305.5 | 313.8 | 305.9 |
| 631.0 | 3881 | -4.7 | -12.7 | 54 | 2.29 | 289 | 33 | 306.2 | 313.6 | 306.6 |
| 600.7 | 4267 | -6.4 | -16.7 | 44 | 1.73 | 295 | 39 | 308.6 | 314.3 | 308.9 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
| 631.0 | 4572 | -4.7 | -12.7 | 54 | 2.29 | 295 | 39 | 306.2 | 313.6 | 306.6 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
| 631.0 | 4572 | -4.7 | -12.7 | 54 | 2.29 | 295 | 39 | 306.2 | 313.6 | 306.6 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
| 631.0 | 4572 | -4.7 | -12.7 | 54 | 2.29 | 295 | 39 | 306.2 | 313.6 | 306.6 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
| 631.0 | 4572 | -4.7 | -12.7 | 54 | 2.29 | 295 | 39 | 306.2 | 313.6 | 306.6 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
| 631.0 | 4572 | -4.7 | -12.7 | 54 | 2.29 | 295 | 39 | 306.2 | 313.6 | 306.6 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
| 631.0 | 4572 | -4.7 | -12.7 | 54 | 2.29 | 295 | 39 | 306.2 | 313.6 | 306.6 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
| 631.0 | 4572 | -4.7 | -12.7 | 54 | 2.29 | 295 | 39 | 306.2 | 313.6 | 306.6 |
| 592.0 | 4381 | -6.9 | -17.9 | 41 | 1.59 | 285 | 30 | 309.3 | 314.6 | 309.6 |
# Parse the data
```
sound_data = StringIO(data_txt)
p,h,T,Td = np.loadtxt(sound_data, usecols=range(0,4), unpack=True)
```

# Create a new figure. The dimensions here give a good aspect ratio
```
fig = plt.figure(figsize=(6.5875, 6.2125))
ax = fig.add_subplot(111, projection='skewx')
```
```
plt.grid(True)
```

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
```
ax.semilogy(T, p, 'r')
ax.semilogy(Td, p, 'g')
```

# An example of a slanted line at constant X
```
l = ax.axvline(0, color='b')
```

# Disables the log-formatting that comes with semilogy
```
ax.yaxis.set_major_formatter(ScalarFormatter())
ax.set_yticks(np.linspace(100,1000,10))
ax.set_ylim(1050,100)
```
```
ax.xaxis.set_major_locator(MultipleLocator(10))
ax.set_xlim(-50,50)
```
```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Illustrate some helper functions for shading regions where a logical mask is True

See :meth:`matplotlib.collections.BrokenBarHCollection.span_where`

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.collections as collections

t = np.arange(0.0, 2, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = 1.2*np.sin(4*np.pi*t)

fig, ax = plt.subplots()
ax.set_title('using span_where')
ax.plot(t, s1, color='black')
ax.axhline(0, color='black', lw=2)

collection = collections.BrokenBarHCollection.span_where(
    t, s1, s2, facecolor='green', edgecolor='black', alpha=0.5)
```

```python
```
```python
ax.add_collection(collection)

collection = collections.BrokenBarHCollection.span_where(
    t, ymin=-1, ymax=0, where=s1<0, facecolor='red', alpha=0.5)
ax.add_collection(collection)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

75.34  api example code: two_scales.py

#!/usr/bin/env python

"""
Demonstrate how to do two plots on the same axes with different left
right scales.
"""

75.34. api example code: two_scales.py 1723
The trick is to use *2 different axes*. Turn the axes rectangular frame off on the 2nd axes to keep it from obscuring the first. Manually set the tick locs and labels as desired. You can use separate matplotlib.ticker formatters and locators as desired since the two axes are independent.

This is achieved in the following example by calling the Axes.twinx() method, which performs this work. See the source of twinx() in axes.py for an example of how to do it for different x scales. (Hint: use the xaxis instance and call tick_bottom and tick_top in place of tick_left and tick_right.)

The twinx and twiny methods are also exposed as pyplot functions.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots()
t = np.arange(0.01, 10.0, 0.01)
s1 = np.exp(t)
ax1.plot(t, s1, 'b-')
ax1.set_xlabel('time (s)')
# Make the y-axis label and tick labels match the line color.
ax1.set_ylabel('exp', color='b')
for tl in ax1.get_yticklabels():
    tl.set_color('b')

ax2 = ax1.twinx()
s2 = np.sin(2*np.pi*t)
ax2.plot(t, s2, 'r-')
ax2.set_ylabel('sin', color='r')
for tl in ax2.get_yticklabels():
    tl.set_color('r')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
You can use the proper typesetting unicode minus (see http://en.wikipedia.org/wiki/Plus_sign#Plus_sign) or the ASCII hypen for minus, which some people prefer. The matplotlibrc param axes.unicode_minus controls the default behavior.

The default is to use the unicode minus

```python
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

matplotlib.rcParams['axes.unicode_minus'] = False
fig, ax = plt.subplots()
ax.plot(10*np.random.randn(100), 10*np.random.randn(100), 'o')
ax.set_title('Using hypen instead of unicode minus')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Use a PNG file as a watermark

```python
from __future__ import print_function
import numpy as np
import matplotlib.cbook as cbook
import matplotlib.image as image
import matplotlib.pyplot as plt

datafile = cbook.get_sample_data('logo2.png', asfileobj=False)
print('loading %s' % datafile)
im = image.imread(datafile)
im[::, :, -1] = 0.5  # set the alpha channel

fig, ax = plt.subplots()

ax.plot(np.random.rand(20), '-o', ms=20, lw=2, alpha=0.7, mfc='orange')
ax.grid()
fig.figimage(im, 10, 10)

plt.show()
```
Use a Text as a watermark

```python
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(np.random.rand(20), '-o', ms=20, lw=2, alpha=0.7, mfc='orange')
ax.grid()

fig.text(0.95, 0.05, 'Property of MPL', fontsize=50, color='gray', ha='right', va='bottom', alpha=0.5)
```

plt.show()

---

75.37. api example code: watermark_text.py
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)
AXES_GRID EXAMPLES

76.1 axes_grid example code: demo_axes Divider.py

![Heatmap Image 1](image1.png)

![Heatmap Image 2](image2.png)
import matplotlib.pyplot as plt

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

def demo_simple_image(ax):
    Z, extent = get_demo_image()

    im = ax.imshow(Z, extent=extent, interpolation="nearest")
    cb = plt.colorbar(im)
    plt.setp(cb.ax.get_yticklabels(), visible=False)

def demo_locatable_axes_hard(fig):
    from mpl_toolkits.axes_grid1 \
      import SubplotDivider, LocatableAxes, Size
    divider = SubplotDivider(fig, 2, 2, 2, aspect=True)
    # axes for image
    ax = LocatableAxes(fig, divider.get_position())
    # axes for colorbar
    ax_cb = LocatableAxes(fig, divider.get_position())
    h = [Size.AxesX(ax),
         Size.Fixed(0.05),
         Size.Fixed(0.2),
         Size.AxesY(ax)]
    v = [Size.AxesY(ax)]
    divider.set_horizontal(h)
    divider.set_vertical(v)
    ax.set_axes_locator(divider.new_locator(nx=0, ny=0))
    ax_cb.set_axes_locator(divider.new_locator(nx=2, ny=0))
    fig.add_axes(ax)
    fig.add_axes(ax_cb)
    ax_cb.axis["left"].toggle(all=False)
    ax_cb.axis["right"].toggle(ticks=True)
    Z, extent = get_demo_image()
```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

def demo_locatable_axes_easy(ax):
    divider = make_axes_locatable(ax)
    ax_cb = divider.new_horizontal(size="5\%", pad=0.05)
    fig1 = ax.get_figure()
    fig1.add_axes(ax_cb)
    Z, extent = get_demo_image()
    im = ax.imshow(Z, extent=extent, interpolation="nearest")
    plt.colorbar(im, cax=ax_cb)
    ax_cb.yaxis.tick_right()
    for tl in ax_cb.get_yticklabels():
        tl.set_visible(False)
    ax_cb.yaxis.tick_right()

def demo_images_side_by_side(ax):
    divider = make_axes_locatable(ax)
    Z, extent = get_demo_image()
    ax2 = divider.new_horizontal(size="100\%", pad=0.05)
    fig1 = ax.get_figure()
    fig1.add_axes(ax2)
    ax.imshow(Z, extent=extent, interpolation="nearest")
    ax2.imshow(Z, extent=extent, interpolation="nearest")
    for tl in ax2.get_yticklabels():
        tl.set_visible(False)

def demo():
    fig1 = plt.figure(1, (6, 6))
    fig1.clf()

    ## PLOT 1
    # simple image & colorbar
    ax = fig1.add_subplot(2, 2, 1)
    demo_simple_image(ax)

    ## PLOT 2
    # image and colorbar whose location is adjusted in the drawing time.
    # a hard way
```

76.1. axes_grid example code: demo_axes_divider.py
demo_locatable_axes_hard(fig1)

## PLOT 3
# image and colorbar whose location is adjusted in the drawing time.
# a easy way

ax = fig1.add_subplot(2, 2, 3)
demo_locatable_axes_easy(ax)

## PLOT 4
# two images side by side with fixed padding.

ax = fig1.add_subplot(2, 2, 4)
demo_images_side_by_side(ax)

plt.draw()
plt.show()

demo()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 76.2 axes_grid example code: demo_axes_grid.py

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

def demo_simple_grid(fig):
    ""
```
A grid of 2x2 images with 0.05 inch pad between images and only
the lower-left axes is labeled.

```python
grid = AxesGrid(fig, 141, # similar to subplot(141)
                nrows_ncols = (2, 2),
                axes_pad = 0.05,
                label_mode = "1",
            )
```

Z, extent = get_demo_image()
for i in range(4):
    im = grid[i].imshow(Z, extent=extent, interpolation="nearest")

# This only affects axes in first column and second row as share_all = False.
grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])

def demo_grid_with_single_cbar(fig):
    # A grid of 2x2 images with a single colorbar
    grid = AxesGrid(fig, 142, # similar to subplot(142)
                    nrows_ncols = (2, 2),
                    axes_pad = 0.0,
                    share_all=True,
                    label_mode = "L",
                    cbar_location = "top",
                    cbar_mode="single",
                )

    Z, extent = get_demo_image()
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest")
    
    #plt.colorbar(im, cax = grid.cbar_axes[0])
grid.cbar_axes[0].colorbar(im)
    for cax in grid.cbar_axes:
        cax.toggle_label(False)

    # This affects all axes as share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])

def demo_grid_with_each_cbar(fig):
    # A grid of 2x2 images. Each image has its own colorbar.
    grid = AxesGrid(F, 143, # similar to subplot(143)
                    nrows_ncols = (2, 2),
                    axes_pad = 0.1,
                )
Z, extent = get_demo_image()
for i in range(4):
    im = grid[i].imshow(Z, extent=extent, interpolation="nearest")
    grid.cbar_axes[i].colorbar(im)

for cax in grid.cbar_axes:
    cax.toggle_label(False)

    # This affects all axes because we set share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])

def demo_grid_with_each_cbar_labelled(fig):
    """
    A grid of 2x2 images. Each image has its own colorbar.
    """
    grid = AxesGrid(F, 144, # similar to subplot(144)
                    nrows_ncols = (2, 2),
                    axes_pad = (0.45, 0.15),
                    label_mode = "1",
                    share_all = True,
                    cbar_location="right",
                    cbar_mode="each",
                    cbar_size="7%",
                    cbar_pad="2%",
                )
    Z, extent = get_demo_image()

    # Use a different colorbar range every time
    limits = ((0, 1), (-2, 2), (-1.7, 1.4), (-1.5, 1))
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                            vmin = limits[i][0], vmax = limits[i][1])
        grid.cbar_axes[i].colorbar(im)

        for i, cax in enumerate(grid.cbar_axes):
            cax.set_yticks((limits[i][0], limits[i][1]))

            # This affects all axes because we set share_all = True.
            grid.axes_llc.set_xticks([-2, 0, 2])
            grid.axes_llc.set_yticks([-2, 0, 2])

    if 1:
        F = plt.figure(1, (10.5, 2.5))
F.subplots_adjust(left=0.05, right=0.95)

demo_simple_grid(F)
demo_grid_with_single_cbar(F)
demo_grid_with_each_cbar(F)
demo_grid_with_each_cbar_labelled(F)

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

76.3 axes_grid example code: demo_axes_grid2.py
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    return z, (-3,4,-4,3)

def add_inner_title(ax, title, loc, size=None, **kwargs):
    from matplotlib.offsetbox import AnchoredText
    from matplotlib.path_effects import withStroke
    if size is None:
        size = dict(size=plt.rcParams['legend.fontsize'])
    at = AnchoredText(title, loc=loc, prop=size,
                      pad=0., borderpad=0.5,
                      frameon=False, **kwargs)
    ax.add_artist(at)
    at.txt._text.set_path_effects([withStroke(foreground="w", linewidth=3)])
    return at

if 1:
    F = plt.figure(1, (6, 6))
    F.clf()
    # prepare images
    Z, extent = get_demo_image()
    ZS = [Z[i::3,:] for i in range(3)]
    extent = extent[0], extent[1]/3., extent[2], extent[3]
    # demo 1 : colorbar at each axes
    grid = ImageGrid(F, 211, # similar to subplot(111)
                     nrows_ncols = (1, 3),
                     direction="row",
                     axes_pad = 0.05,
                     add_all=True,
                     label_mode = "1",
                     share_all = True,
                     char_location="top",
                     char_mode="each",
                     char_size="7%",
                     char_pad="1%",
                     )
    for ax, z in zip(grid, ZS):
        im = ax.imshow(z, origin="lower", extent=extent, interpolation="nearest")
        ax.cax.colorbar(im)
for ax, im_title in zip(grid, ["Image 1", "Image 2", "Image 3"]):
    t = add_inner_title(ax, im_title, loc=3)
    t.patch.set_alpha(0.5)

for ax, z in zip(grid, ZS):
    ax.cax.toggle_label(True)
    #axis = ax.cax.axis[ax.cax.orientation]
    #axis.label.set_text("counts s^{-1}\$")
    #axis.label.set_size(10)
    #axis.major_ticklabels.set_size(6)

    # changing the colorbar ticks
    grid[0].cax.set_xticks([-1, 0, 1])
    grid[2].cax.set_xticks([-1, 0, 1])

    grid[0].set_xticks([-2, 0])
    grid[0].set_yticks([-2, 0, 2])

# demo 2 : shared colorbar

grid2 = ImageGrid(F, 212,
    nrows_ncols = (1, 3),
    direction="row",
    axes_pad = 0.05,
    add_all=True,
    label_mode = "1",
    share_all = True,
    cbar_location="right",
    cbar_mode="single",
    cbar_size="10%",
    cbar_pad=0.05,
)

grid2[0].set_xlabel("X")
grid2[0].set_ylabel("Y")

vmax, vmin = np.max(ZS), np.min(ZS)
import matplotlib.colors
norm = matplotlib.colors.Normalize(vmax=vmax, vmin=vmin)

for ax, z in zip(grid2, ZS):
    im = ax.imshow(z, norm=norm,
                    origin="lower", extent=extent,
                    interpolation="nearest")

    ax.cax.colorbar(im)
    ax.cax.toggle_label(True)

for ax, im_title in zip(grid2, ["(a)", "(b)", "(c)"]):
    t = add_inner_title(ax, im_title, loc=2)
    t.patch.set_ec("none")
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.axes_divider import HBoxDivider
import matplotlib.ticker as ticker
import mpl_toolkits.axes_grid1.axes_size as Size

def make_heights_equal(fig, rect, ax1, ax2, pad):
    # pad in inches
    h1, v1 = Size.AxesX(ax1), Size.AxesY(ax1)
    h2, v2 = Size.AxesX(ax2), Size.AxesY(ax2)
pad_v = Size.Scaled(1)
pad_h = Size.Fixed(pad)

my_divider = HBoxDivider(fig, rect,
    horizontal=[h1, pad_h, h2],
    vertical=[v1, pad_v, v2])

ax1.set_axes_locator(my_divider.new_locator(0))
ax2.set_axes_locator(my_divider.new_locator(2))

if __name__ == "__main__":
    arr1 = np.arange(20).reshape((4,5))
    arr2 = np.arange(20).reshape((5,4))

    fig, (ax1, ax2) = plt.subplots(1,2)
    ax1.imshow(arr1, interpolation="nearest")
    ax2.imshow(arr2, interpolation="nearest")

    rect = 111 # subplot param for combined axes
    make_heights_equal(fig, rect, ax1, ax2, pad=0.5) # pad in inches

    for ax in [ax1, ax2]:
        ax.locator_params(nbins=4)

    # annotate
    ax3 = plt.axes([0.5, 0.5, 0.001, 0.001], frameon=False)
    ax3.xaxis.set_visible(False)
    ax3.yaxis.set_visible(False)
    ax3.annotate("Location of two axes are adjusted\nso that they have equal heights\nwhile maintaining their aspect ratios", (0.5, 0.5),
                xycoords="axes fraction", va="center", ha="center",
                bbox=dict(boxstyle="round, pad=1", fc="w"))

    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
76.5 axes_grid example code: demo_axes_rgb.py
import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.axes_rgb import make_rgb_axes, RGBAxes

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

def get_rgb():
    Z, extent = get_demo_image()

    Z[Z<0] = 0.
    Z = Z/Z.max()

    R = Z[:13,:13]
    G = Z[2:,2:]
    B = Z[:13,2:]

    return R, G, B
```python
def make_cube(r, g, b):
    ny, nx = r.shape
    R = np.zeros([ny, nx, 3], dtype="d")
    R[:,:,0] = r
    G = np.zeros_like(R)
    G[:,:,1] = g
    B = np.zeros_like(R)
    B[:,:,2] = b
    RGB = R + G + B
    return R, G, B, RGB

def demo_rgb():
    fig, ax = plt.subplots()
    ax_r, ax_g, ax_b = make_rgb_axes(ax, pad=0.02)
    #fig.add_axes(ax_r)
    #fig.add_axes(ax_g)
    #fig.add_axes(ax_b)
    
    r, g, b = get_rgb()
    im_r, im_g, im_b, im_rgb = make_cube(r, g, b)
    kwargs = dict(origin="lower", interpolation="nearest")
    ax.imshow(im_rgb, **kwargs)
    ax_r.imshow(im_r, **kwargs)
    ax_g.imshow(im_g, **kwargs)
    ax_b.imshow(im_b, **kwargs)

def demo_rgb2():
    fig = plt.figure(2)
    ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8], pad=0.0)
    #fig.add_axes(ax)
    #ax.add_RGB_to_figure()
    
    r, g, b = get_rgb()
    kwargs = dict(origin="lower", interpolation="nearest")
    ax.imshow_rgb(r, g, b, **kwargs)
    ax.RGB.set_xlim(0., 9.5)
    ax.RGB.set_ylim(0.9, 10.6)
    
    for ax1 in [ax.RGB, ax.R, ax.G, ax.B]:
        for spl in ax1.spines.values():
            spl.set_color("w")
        for tick in ax1.xaxis.get_major_ticks() + ax1.yaxis.get_major_ticks():
            tick.tick1line.set_mec("w")
            tick.tick2line.set_mec("w")
```

return ax

demo_rgb()
ax = demo_rgb2()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

76.6 axes_grid example code: demo_axisline_style.py

```python
from mpl_toolkits.axes_grid.axislines import SubplotZero
import matplotlib.pyplot as plt
import numpy as np

if 1:
    fig = plt.figure(1)
    ax = SubplotZero(fig, 111)
    fig.add_subplot(ax)

    for direction in ["xzero", "yzero"]:
```

76.6. axes_grid example code: demo_axisline_style.py 1743
```python
ax.axis[direction].set_axisline_style("-|>")
ax.axis[direction].set_visible(True)

for direction in ["left", "right", "bottom", "top"]:
    ax.axis[direction].set_visible(False)

x = np.linspace(-0.5, 1., 100)
ax.plot(x, np.sin(x*np.pi))

plt.show()

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import inset_axes

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=[6, 3])

axins1 = inset_axes(ax1,
                     width="50%", # width = 10% of parent_bbox width
                     height="5%", # height : 50%
                     loc=1)

im1=ax1.imshow([[1,2],[2, 3]])
plt.colorbar(im1, cax=axins1, orientation="horizontal", ticks=[1,2,3])
axins1.xaxis.set_ticks_position("bottom")

im1=ax1.imshow([[1,2],[2, 3]])
plt.colorbar(im1, cax=axins1, orientation="horizontal", ticks=[1,2,3])
axins1.xaxis.set_ticks_position("bottom")
```

---

76.7 axes_grid example code: demo_colorbar_with_inset_locator.py
### 76.8 axes_grid example code: demo_curvelinear_grid.py

```python
import numpy as np
# from matplotlib.path import Path

import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

# Controlling the placement of the inset axes is basically same as that # of the legend. You may want to play with the borderpad value and # the bbox_to_anchor coordinate.

im=ax2.imshow([[1, 2], [2, 3]])
plt.colorbar(im, cax=axins, ticks=[1, 2, 3])

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
from mpl_toolkits.axisartist.grid_helper_curvelinear import GridHelperCurveLinear
from mpl_toolkits.axisartist import Subplot
from mpl_toolkits.axisartist import SubplotHost,
ParasiteAxesAuxTrans

def curvelinear_test1(fig):
    ""
    grid for custom transform.
    ""

def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

def inv_tr(x,y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y+x

grid_helper = GridHelperCurveLinear((tr, inv_tr))

ax1 = Subplot(fig, 1, 2, 1, grid_helper=grid_helper)
# ax1 will have a ticks and gridlines defined by the given
# transform (+ transData of the Axes). Note that the transform of
# the Axes itself (i.e., transData) is not affected by the given
# transform.
fig.add_subplot(ax1)

xx, yy = tr([3, 6], [5.0, 10.])
ax1.plot(xx, yy)

ax1.set_aspect(1.)
ax1.set_xlim(0, 10.)
ax1.set_ylim(0, 10.)

ax1.axis['t']=ax1.new_floating_axis(0, 3.)
ax1.axis['t2']=ax1.new_floating_axis(1, 7.)
ax1.grid(True)

import mpl_toolkits.axisartist.angle_helper as angle_helper
from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D

def curvelinear_test2(fig):
    ""
    polar projection, but in a rectangular box.
    ""
# PolarAxes.PolarTransform takes radian. However, we want our coordinate
# system in degree
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

# polar projection, which involves cycle, and also has limits in
# its coordinates, needs a special method to find the extremes
# (min, max of the coordinate within the view).

# 20, 20 : number of sampling points along x, y direction
extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
    lon_cycle = 360,
    lat_cycle = None,
    lon_minmax = None,
    lat_minmax = (0, np.inf),
)

grid_locator1 = angle_helper.LocatorDMS(12)
# Find a grid values appropriate for the coordinate (degree,
# minute, second).

tick_formatter1 = angle_helper.FormatterDMS()
# And also uses an appropriate formatter. Note that, the
# acceptable Locator and Formatter class is a bit different than
# that of mpl's, and you cannot directly use mpl's Locator and
# Formatter here (but may be possible in the future).

grid_helper = GridHelperCurveLinear(tr,
    extreme_finder=extreme_finder,
    grid_locator1=grid_locator1,
    tick_formatter1=tick_formatter1
)

ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

# make ticklabels of right and top axis visible.
ax1.axis["right"].major_ticklabels.set_visible(True)
ax1.axis["top"].major_ticklabels.set_visible(True)

# let right axis shows ticklabels for 1st coordinate (angle)
ax1.axis["right"].get_helper().nth_coord_ticks=0
# let bottom axis shows ticklabels for 2nd coordinate (radius)
ax1.axis["bottom"].get_helper().nth_coord_ticks=1

fig.add_subplot(ax1)

# A parasite axes with given transform
ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anthing you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)
intp = cbook.simple_linear_interpolation

76.8. axes_grid example code: demo_curvelinear_grid.py
ax2.plot(intp(np.array([0, 30]), 50),
         intp(np.array([10., 10.]), 50))
ax1.set_aspect(1.)
ax1.set_xlim(-5, 12)
ax1.set_ylim(-5, 10)
ax1.grid(True)

if 1:
    fig = plt.figure(1, figsize=(7, 4))
    fig.clf()
    curvelinear_test1(fig)
    curvelinear_test2(fig)
    plt.draw()
    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

76.9 axes_grid example code: demo_curvelinear_grid2.py

import numpy as np
# from matplotlib.path import Path
import matplotlib.pyplot as plt

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid.grid_helper_curvelinear import GridHelperCurveLinear
from mpl_toolkits.axes_grid.axislines import Subplot
import mpl_toolkits.axes_grid.angle_helper as angle_helper
def curvelinear_test1(fig):
    """
    grid for custom transform.
    """

def tr(x, y):
    sgn = np.sign(x)
    x, y = np.abs(np.asarray(x)), np.asarray(y)
    return sgn*x**.5, y

def inv_tr(x,y):
    sgn = np.sign(x)
    x, y = np.asarray(x), np.asarray(y)
    return sgn*x**2, y

extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                  lon_cycle = None,
                                                  lat_cycle = None,
                                                  lon_minmax = None,
                                                  lat_minmax = None,
                                                  )

grid_helper = GridHelperCurveLinear((tr, inv_tr),
                                     extreme_finder=extreme_finder)

ax1 = Subplot(fig, 111, grid_helper=grid_helper)
# ax1 will have a ticks and gridlines defined by the given
# transform (+ transData of the Axes). Note that the transform of
# the Axes itself (i.e., transData) is not affected by the given
# transform.
fig.add_subplot(ax1)

ax1.imshow(np.arange(25).reshape(5,5),
           vmax = 50, cmap=plt.cm.gray_r,
           interpolation="nearest",
           origin="lower")

# tick density
grid_helper.grid_finder.grid_locator1._nbins = 6
grid_helper.grid_finder.grid_locator2._nbins = 6

if 1:
    fig = plt.figure(1, figsize=(7, 4))
    fig.clf()
Matplotlib, Release 1.4.3

curvelinear_test1(fig)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

76.10 axes_grid example code: demo_edge_colorbar.py

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

def demo_bottom_cbar(fig):
    
    A grid of 2x2 images with a colorbar for each column.
    
    grid = AxesGrid(fig, 121, # similar to subplot(321)
                    nrows_ncols = (2, 2),
                    axes_pad = 0.10,
                    share_all=True,
                    label_mode = "1",
                    cbar_location = "bottom",
                    cbar_mode="edge",
                    cbar_pad = 0.25,
                    cbar_size = "15%",
                    direction="column"
                    )
Z, extent = get_demo_image()
cmaps = [plt.get_cmap("autumn"), plt.get_cmap("summer")]
for i in range(4):
    im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                        cmap=cmaps[i//2])
    if i % 2:
        cbar = grid.cbar_axes[i//2].colorbar(im)

for cax in grid.cbar_axes:
    cax.toggle_label(True)
    cax.axis[cax.orientation].set_label("Bar")

# This affects all axes as share_all = True.
grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])

def demo_right_cbar(fig):
    
    A grid of 2x2 images. Each row has its own colorbar.

    grid = AxesGrid(F, 122, # similar to subplot(122)
                    nrows_ncols = (2, 2),
                    axes_pad = 0.10,
                    label_mode = "1",
                    share_all = True,
                    cbar_location="right",
                    cbar_mode="edge",
                    cbar_size="7%",
                    cbar_pad="2%",
                    )

    Z, extent = get_demo_image()
cmaps = [plt.get_cmap("spring"), plt.get_cmap("winter")]
for i in range(4):
    im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                        cmap=cmaps[i//2])
    if i % 2:
        grid.cbar_axes[i//2].colorbar(im)

for cax in grid.cbar_axes:
    cax.toggle_label(True)
    cax.axis[cax.orientation].set_label('Foo')

    # This affects all axes because we set share_all = True.
grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])

if 1:
    F = plt.figure(1, (5.5, 2.5))
```python
from matplotlib.transforms import Affine2D
import mpl_toolkits.axisartist.floating_axes as floating_axes
import numpy as np
import mpl_toolkits.axisartist.angle_helper as angle_helper
from matplotlib.projections import PolarAxes
from mpl_toolkits.axisartist.grid_finder import FixedLocator, MaxNLocator, 
    DictFormatter

def setup_axes1(fig, rect):
    """
    A simple one.
    """
    tr = Affine2D().scale(2, 1).rotate_deg(30)

    grid_helper = floating_axes.GridHelperCurveLinear(tr, extremes=(0, 4, 0, 4))
```
ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
fig.add_subplot(ax1)

aux_ax = ax1.get_aux_axes(tr)

grid_helper.grid_finder.grid_locator1._nbins = 4
grid_helper.grid_finder.grid_locator2._nbins = 4

return ax1, aux_ax

def setup_axes2(fig, rect):
    
    # With custom locator and formatter.
    # Note that the extreme values are swapped.
    
    #tr_scale = Affine2D().scale(np.pi/180., 1.)
    tr = PolarAxes.PolarTransform()
    pi = np.pi
    angle_ticks = [((0, r"$0$"),
                  (.25*pi, r"$\frac{1}{4}\pi$"),
                  (.5*pi, r"$\frac{1}{2}\pi$"))]
    grid_locator1 = FixedLocator([v for v, s in angle_ticks])
    tick_formatter1 = DictFormatter(dict(angle_ticks))
    grid_locator2 = MaxNLocator(2)
    grid_helper = floating_axes.GridHelperCurveLinear(tr,
                                                      extremes=(.5*pi, 0, 2, 1),
                                                      grid_locator1=grid_locator1,
                                                      grid_locator2=grid_locator2,
                                                      tick_formatter1=tick_formatter1,
                                                      tick_formatter2=None,
                                                      )

    ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
    fig.add_subplot(ax1)

    # create a parasite axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch  # for aux_ax to have a clip path as in ax
    ax1.patch.zorder=0.9  # but this has a side effect that the patch is 
                          # drawn twice, and possibly over some other
                          # artists. So, we decrease the zorder a bit to
                          # prevent this.

    return ax1, aux_ax
def setup_axes3(fig, rect):
    
    
    Sometimes, things like axis_direction need to be adjusted.
    
    
    # rotate a bit for better orientation
    tr_rotate = Affine2D().translate(-95, 0)

    # scale degree to radians
    tr_scale = Affine2D().scale(np.pi/180, 1.)

    tr = tr_rotate + tr_scale + PolarAxes.PolarTransform()

    grid_locator1 = angle_helper.LocatorHMS(4)
    tick_formatter1 = angle_helper.FormatterHMS()

    grid_locator2 = MaxNLocator(3)

    ra0, ra1 = 8.15, 14.15
    cz0, cz1 = 0, 14000
    grid_helper = floating_axes.GridHelperCurveLinear(tr,
           extremes=(ra0, ra1, cz0, cz1),
           grid_locator1=grid_locator1,
           grid_locator2=grid_locator2,
           tick_formatter1=tick_formatter1,
           tick_formatter2=None,
           )

    ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
    fig.add_subplot(ax1)

    # adjust axis
    ax1.axis["left"].set_axis_direction("bottom")
    ax1.axis["right"].set_axis_direction("top")

    ax1.axis["bottom"].set_visible(False)
    ax1.axis["top"].set_axis_direction("bottom")
    ax1.axis["top"].toggle(ticklabels=True, label=True)
    ax1.axis["top"].major_ticklabels.set_axis_direction("top")
    ax1.axis["top"].label.set_axis_direction("top")

    ax1.axis["left"].label.set_text(r"cz \ [km^{-1}]")
    ax1.axis["top"].label.set_text(r"$\alpha_{1950}$")

    # create a parasite axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch  # for aux_ax to have a clip path as in ax
    ax1.patch.zorder=0.9  # but this has a side effect that the patch is
    # drawn twice, and possibly over some other
    # artists. So, we decrease the zorder a bit to
    # prevent this.
return ax1, aux_ax

if 1:
    import matplotlib.pyplot as plt
    fig = plt.figure(1, figsize=(8, 4))
    fig.subplots_adjust(wspace=0.3, left=0.05, right=0.95)
    ax1, aux_ax2 = setup_axes1(fig, 131)
    aux_ax2.bar([0, 1, 2, 3], [3, 2, 1, 3])
    #theta = np.random.rand(10) #*.5*np.pi
    #radius = np.random.rand(10) #+1.
    #aux_ax1.scatter(theta, radius)

    ax2, aux_ax2 = setup_axes2(fig, 132)
    theta = np.random.rand(10)*.5*np.pi
    radius = np.random.rand(10)+1.
    aux_ax2.scatter(theta, radius)

    ax3, aux_ax3 = setup_axes3(fig, 133)
    theta = (8 + np.random.rand(10)*(14-8))*15.  # in degrees
    radius = np.random.rand(10)*14000.
    aux_ax3.scatter(theta, radius)

    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
76.12 axes_grid example code: demo_floating_axis.py

```python
def curvelinear_test2(fig):
    """
    polar projection, but in a rectangular box.
    """
    global ax1
    import numpy as np
    import mpl_toolkits.axisartist.angle_helper as angle_helper
    from matplotlib.projections import PolarAxes
    from matplotlib.transforms import Affine2D
    from mpl_toolkits.axisartist import SubplotHost
    from mpl_toolkits.axisartist import GridHelperCurveLinear
```
# see demo_curvelinear_grid.py for details
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
    lon_cycle = 360,
    lat_cycle = None,
    lon_minmax = None,
    lat_minmax = (0, np.inf),
)

grid_locator1 = angle_helper.LocatorDMS(12)
tick_formatter1 = angle_helper.FormatterDMS()

grid_helper = GridHelperCurveLinear(tr,
    extreme_finder=extreme_finder,
    grid_locator1=grid_locator1,
    tick_formatter1=tick_formatter1
)

ax1 = SubplotHost(fig, 1, 1, 1, grid_helper=grid_helper)
fig.add_subplot(ax1)

# Now creates floating axis

#grid_helper = ax1.get_grid_helper()
# floating axis whose first coordinate (theta) is fixed at 60
ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^\circ$"
axis.label.set_visible(True)

# floating axis whose second coordinate (r) is fixed at 6
ax1.axis["lon"] = axis = ax1.new_floating_axis(1, 6)
axis.label.set_text(r"$r = 6$"

ax1.set_aspect(1.)
ax1.set_xlim(-5, 12)
ax1.set_ylim(-5, 10)

ax1.grid(True)

import matplotlib.pyplot as plt
fig = plt.figure(1, figsize=(5, 5))
fig.clf()
curvelinear_test2(fig)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
fig = plt.figure(1)
grid1 = ImageGrid(fig, 121, (2,2), axes_pad=0.1,
                 aspect=True, share_all=True)
for i in [0, 1]:
    grid1[i].set_aspect(2)

grid2 = ImageGrid(fig, 122, (2,2), axes_pad=0.1,
                 aspect=True, share_all=True)
for i in [1, 3]:
    grid2[i].set_aspect(2)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.axes_grid1 import host_subplot
import mpl_toolkits.axisartist as AA
import matplotlib.pyplot as plt

if 1:
    host = host_subplot(111, axes_class=AA.Axes)
    plt.subplots_adjust(right=0.75)
    par1 = host.twinx()
    par2 = host.twinx()

    offset = 60
    new_fixed_axis = par2.get_grid_helper().new_fixed_axis
    par2.axis['right'] = new_fixed_axis(loc="right",
                                        axes=par2,
                                        offset=(offset, 0))

    par2.axis['right'].toggle(all=True)
```python
host.set_xlim(0, 2)
host.set_ylim(0, 2)

host.set_xlabel("Distance")
host.set_ylabel("Density")
par1.set_ylabel("Temperature")
par2.set_ylabel("Velocity")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par1.plot([0, 1, 2], [0, 3, 2], label="Temperature")
p3, = par2.plot([0, 1, 2], [50, 30, 15], label="Velocity")

par1.set_ylim(0, 4)
par2.set_ylim(1, 65)

host.legend()

host.axis["left"].label.set_color(p1.get_color())
par1.axis["right"].label.set_color(p2.get_color())
par2.axis["right"].label.set_color(p3.get_color())

plt.draw()
plt.show()

# plt.savefig("Test")
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

76.15 axes_grid example code: inset_locator_demo.py

![Inset Locator Demo](image)

76.15 axes_grid example code: inset_locator_demo.py
```python
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import inset_axes, zoomed_inset_axes
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar

def add_sizebar(ax, size):
    asb = AnchoredSizeBar(ax.transData, 
                          size, 
                          str(size), 
                          loc=8, 
                          pad=0.1, borderpad=0.5, sep=5, 
                          frameon=False)
    ax.add_artist(asb)

fig, (ax, ax2) = plt.subplots(1, 2, figsize=[5.5, 3])

# first subplot
ax.set_aspect(1.)
axins = inset_axes(ax, 
                   width="30\%", # width = 30\% of parent_bbox
                   height=1., # height : 1 inch
                   loc=3)
plt.xticks(visible=False)
plt.yticks(visible=False)

# second subplot
ax2.set_aspect(1.)
axins = zoomed_inset_axes(ax2, 0.5, loc=1) # zoom = 0.5
plt.xticks(visible=False)
plt.yticks(visible=False)

add_sizebar(ax2, 0.5)
add_sizebar(axins, 0.5)

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
76.16 axes_grid example code: inset_locator_demo2.py

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes,
    mark_inset
import numpy as np

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

fig, ax = plt.subplots(figsize=[5, 4])

# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30+ny, 30:30+nx] = Z

# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest",)
```

axins = zoomed_inset_axes(ax, 6, loc=1)  # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest",
origin="lower")

# sub region of the original image
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)

plt.xticks(visible=False)
plt.yticks(visible=False)

# draw a bbox of the region of the inset axes in the parent axes and
# connecting lines between the bbox and the inset axes area
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
76.17 axes_grid example code: make_room_for_ylabel_using_axesgrid.py
from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.axes_grid1.axes_divider import make_axes_area_auto_adjustable

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    def ex1():
        plt.figure(1)
        ax = plt.axes([0,0,1,1])
        # ax = plt.subplot(111)
        ax.set_yticks([0.5])
        ax.set_yticklabels(['very long label'])
        make_axes_area_auto_adjustable(ax)

    def ex2():
        plt.figure(2)
        ax1 = plt.axes([0,0,1,0.5])
        ax2 = plt.axes([0,0.5,1,0.5])
ax1.set_yticks([0.5])
ax1.set_yticklabels(['very long label'])
ax1.set_ylabel("Y label")

ax2.set_title("Title")

make_axes_area_auto_adjustable(ax1, pad=0.1, use_axes=[ax1, ax2])
make_axes_area_auto_adjustable(ax2, pad=0.1, use_axes=[ax1, ax2])

def ex3():
    fig = plt.figure(3)
    ax1 = plt.axes([0,0,1,1])
    divider = make_axes_locatable(ax1)

    ax2 = divider.new_horizontal("100\%", pad=0.3, sharey=ax1)
    ax2.tick_params(labelleft="off")
    fig.add_axes(ax2)

    divider.add_auto_adjustable_area(use_axes=[ax1], pad=0.1,
                                    adjust_dirs=["left"])
    divider.add_auto_adjustable_area(use_axes=[ax2], pad=0.1,
                                    adjust_dirs=["right"])
    divider.add_auto_adjustable_area(use_axes=[ax1, ax2], pad=0.1,
                                    adjust_dirs=["top", "bottom"])

    ax1.set_yticks([0.5])
    ax1.set_yticklabels(['very long label'])

    ax2.set_title("Title")
    ax2.set_xlabel("X - Label")

    ex1()
    ex2()
    ex3()

    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
76.18 axes_grid example code: parasite_simple2.py

import matplotlib.transforms as mtransforms
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.parasite_axes import SubplotHost

obs = [['01_S1', 3.88, 0.14, 1970, 63],
       ['01_S4', 5.6, 0.82, 1622, 150],
       ['02_S1', 2.4, 0.54, 1570, 40],
       ['03_S1', 4.1, 0.62, 2380, 170]]

fig = plt.figure()
ax_kms = SubplotHost(fig, 1,1,1, aspect=1.)

# angular proper motion("/yr) to linear velocity(km/s) at distance=2.3kpc
pm_to_kms = 1./206265.*2300*3.085e18/3.15e7/1.e5

aux_trans = mtransforms.Affine2D().scale(pm_to_kms, 1.)
ax_pm = ax_kms.twin(aux_trans)
ax_pm.set_viewlim_mode("transform")

fig.add_subplot(ax_kms)
for n, ds, dse, w, we in obs:
    time = ((2007+(10. + 4/30.)/12)-1988.5)
    v = ds / time * pm_to_kms
    ve = dse / time * pm_to_kms
    ax_kms.errorbar([v], [w], xerr=[ve], yerr=[we], color="k")

ax_kms.axis["bottom"].set_label("Linear velocity at 2.3 kpc [km/s]")
ax_kms.axis["left"].set_label("FWHM [km/s]")
ax_pm.axis["top"].set_label("Proper Motion ["^"/yr]")
ax_pm.axis["top"].label.set_visible(True)
ax_pm.axis["right"].major_ticklabels.set_visible(False)

ax_kms.set_xlim(950, 3700)
ax_kms.set_ylim(950, 3100)
# xlim and ylim of ax_pms will be automatically adjusted.

plt.draw()
plt.show()
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)

fig, axScatter = plt.subplots(figsize=(5.5,5.5))

# the scatter plot:
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes
# The first argument of the new_vertical(new_horizontal) method is
# the height (width) of the axes to be created in inches.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", 1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", 1.2, pad=0.1, sharey=axScatter)

# make some labels invisible
plt.setp(axHistx.get_xticklabels() + axHisty.get_yticklabels(),
         visible=False)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max([np.max(np.fabs(x)), np.max(np.fabs(y))])
lim = (int(xymax/binwidth) + 1) * binwidth

bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

# the xaxis of axHistx and yaxis of axHisty are shared with axScatter,
# thus there is no need to manually adjust the xlim and ylim of these
# axis.

#axHistx.axis['bottom'].major_ticklabels.set_visible(False)
for tl in axHistx.get_xticklabels():
    tl.set_visible(False)
axHistx.set_yticks([0, 50, 100])

#axHisty.axis['left'].major_ticklabels.set_visible(False)
for tl in axHisty.get_yticklabels():
    tl.set_visible(False)
axHisty.set_xticks([0, 50, 100])

plt.draw()
plt.show()
import matplotlib.pyplot as plt

def draw_text(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredText
    at = AnchoredText("Figure 1a",
                      loc=2, prop=dict(size=8), frameon=True,
                      )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
                       loc=3, prop=dict(size=8), frameon=True,
                       bbox_to_anchor=(0., 1.),
                       bbox_transform=ax.transAxes
                      )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at2)

def draw_circle(ax):
    # circle in the canvas coordinate
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
    from matplotlib.patches import Circle
ada = AnchoredDrawingArea(20, 20, 0, 0,
    loc=1, pad=0., frameon=False)
p = Circle((10, 10), 10)
ada.da.add_artist(p)
ax.add_artist(ada)

def draw_ellipse(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                          loc=3, pad=0.5, borderpad=0.4, frameon=True)

    ax.add_artist(ae)

def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData, 0.1,
                          r"1$^{{\prime}}$",
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)

    ax.add_artist(asb)

if 1:
    ax = plt.gca()
    ax.set_aspect(1.)

draw_text(ax)
draw_circle(ax)
draw_ellipse(ax)
draw_sizebar(ax)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
76.21 axes_grid example code: simple_axesgrid.py

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
nrows_ncols = (2, 2),  # creates 2x2 grid of axes
        axes_pad=0.1,  # pad between axes in inch.
)

for i in range(4):
    grid[i].imshow(im)  # The AxesGrid object work as a list of axes.

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

F = plt.figure(1, (5.5, 3.5))
grid = ImageGrid(F, 111,  # similar to subplot(111)
nrows_ncols = (1, 3),
axes_pad = 0.1,
add_all=True,
label_mode = "L",
)

Z, extent = get_demo_image()  # demo image

im1=Z
im2=Z[:,:10]
im3=Z[:,10:]
vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = grid[i]
    ax.imshow(im, origin="lower", vmin=vmin, vmax=vmax, interpolation="nearest")

76.22. axes_grid example code: simple_axesgrid2.py 1775
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import host_subplot
import mpl_toolkits.axisartist as AA
import numpy as np

ax = host_subplot(111, axes_class=AA.Axes)
x = np.arange(0, 2*np.pi, 0.01)
ax.plot(x, np.sin(x))

ax2 = ax.twin()
ax2.set_xticks([0, 0.5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
ax2.set_xticklabels(['$0$', r'$\frac{1}{2}\pi$', r'$\pi$', r'$\frac{3}{2}\pi$', r'$2\pi$'])

ax.axis['right'].major_ticklabels.set_visible(False)

plt.draw()
plt.show()
plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
77.1 color example code: color_cycle_demo.py

```python
import matplotlib.pyplot as plt
import numpy as np

# Demo of custom color-cycle settings to control colors for multi-line plots.

def color_cycle_demo():
    # Set default color cycle to rgby
    plt.rc('axes', color_cycle=['r','g','b','y'])
    x = np.linspace(0, 50, 1000)
    y1 = np.sin(x)
    y2 = np.cos(x)
    y3 = np.sin(x) * np.cos(x)
    plt.plot(x, y1, label='sine', color='r', lw=2)
    plt.plot(x, y2, label='cosine', color='g', lw=2)
    plt.plot(x, y3, label='sine cosine', color='b', lw=2)
    plt.legend()
    plt.title('Set default color cycle to rgby')
    plt.show()

# Set axes color cycle to cmyk
plt.rc('axes', color_cycle=['c','m','y','k'])
plt.plot(x, y1, label='sine', color='r', lw=2)
plt.plot(x, y2, label='cosine', color='g', lw=2)
plt.plot(x, y3, label='sine cosine', color='b', lw=2)
plt.legend()
plt.title('Set axes color cycle to cmyk')
plt.show()
```

Demo of custom color-cycle settings to control colors for multi-line plots.

This example demonstrates two different APIs:

1. Setting the default rc-parameter specifying the color cycle. This affects all subsequent plots.
2. Setting the color cycle for a specific axes. This only affects a single
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi)
offsets = np.linspace(0, 2 * np.pi, 4, endpoint=False)
# Create array with shifted-sine curve along each column
yy = np.transpose([np.sin(x + phi) for phi in offsets])

plt.rc('lines', linewidth=4)
fig, (ax0, ax1) = plt.subplots(nrows=2)

plt.rc('axes', color_cycle=['r', 'g', 'b', 'y'])
ax0.plot(yy)
ax0.set_title('Set default color cycle to rgbvy')

ax1.set_color_cycle(['c', 'm', 'y', 'k'])
ax1.plot(yy)
ax1.set_title('Set axes color cycle to cmyk')

# Tweak spacing between subplots to prevent labels from overlapping
plt.subplots_adjust(hspace=0.3)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
77.2 color example code: colormaps_reference.py

Sequential colormaps

- Blues
- BuGn
- BuPu
- GnBu
- Greens
- Greys
- Oranges
- OrRd
- PuBu
- PuBuGn
- PuRd
- Purples
- RdPu
- Reds
- YIgGn
- YIgGnBu
- YlOrBr
- YlOrRd
Sequential (2) colormaps

afmhot
autumn
bone
cool
copper
gist_heat
gray
hot
pink
spring
summer
winter
Diverging colormaps

- BrBG
- bwr
- coolwarm
- PiYG
- PRGn
- PuOr
- RdBu
- RdGy
- RdYlBu
- RdYlGn
- Spectral
- seismic
Qualitative colormaps

- Accent
- Dark2
- Paired
- Pastel1
- Pastel2
- Set1
- Set2
- Set3
Reference for colormaps included with Matplotlib.

This reference example shows all colormaps included with Matplotlib. Note that any colormap listed here can be reversed by appending "_r" (e.g., "pink_r"). These colormaps are divided into the following categories:

Sequential:
These colormaps are approximately monochromatic colormaps varying smoothly between two color tones---usually from low saturation (e.g. white) to high saturation (e.g. a bright blue). Sequential colormaps are ideal for representing most scientific data since they show a clear progression from low-to-high values.

Diverging:
These colormaps have a median value (usually light in color) and vary smoothly to two different color tones at high and low values. Diverging colormaps are ideal when your data has a median value that is significant (e.g. 0, such that positive and negative values are represented by different colors of the colormap).

Qualitative:
These colormaps vary rapidly in color. Qualitative colormaps are useful for choosing a set of discrete colors. For example:
Matplotlib, Release 1.4.3

gives a list of RGB colors that are good for plotting a series of lines on a dark background.

Miscellaneous:
Colormaps that don't fit into the categories above.

""
import numpy as np
import matplotlib.pyplot as plt

color_list = plt.cm.Set3(np.linspace(0, 1, 12))

gives a list of RGB colors that are good for plotting a series of lines on a dark background.

Miscellaneous:
Colormaps that don’t fit into the categories above.

""
import numpy as np
import matplotlib.pyplot as plt

('Sequential (2)', ['afmhot', 'autumn', 'bone', 'cool', 'copper', 'gist_heat', 'gray', 'hot', 'pink', 'spring', 'summer', 'winter'])),
('Diverging', ['BrBG', 'bwr', 'coolwarm', 'PiYG', 'PRGn', 'PuOr', 'RdBu', 'RdGy', 'RdYlBu', 'RdYlGn', 'Spectral', 'seismic'])),
('Qualitative', ['Accent', 'Dark2', 'Paired', 'Pastel1', 'Pastel2', 'Set1', 'Set2', 'Set3'])),
('Miscellaneous', ['gist_earth', 'terrain', 'ocean', 'gist_stern', 'brg', 'CMRmap', 'cubehelix', 'gnuplot', 'gnuplot2', 'gist_nice', 'npy_spectral', 'jet', 'rainbow', 'gist_rainbow', 'hsv', 'flag', 'prism'])]

def plot_color_gradients(cmap_category, cmap_list):
    fig, axes = plt.subplots(nrows=nrows)
    fig.subplots_adjust(top=0.95, bottom=0.01, left=0.2, right=0.99)
    axes[0].set_title(cmap_category + ' colormaps', fontsize=14)
    for ax, name in zip(axes, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=plt.get_cmap(name))
        pos = list(ax.get_position().bounds)
        x_text = pos[0] - 0.01
        fig.text(x_text, y_text, name, va='center', ha='right', fontsize=10)

    # Turn off "all" ticks & spines, not just the ones with colormaps.
    for ax in axes:
        ax.set_axis_off()
Matplotlib, Release 1.4.3

plot_color_gradients(cmap_category, cmap_list)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

77.3 color example code: named_colors.py

Visualization of named colors.

Simple plot example with the named colors and its visual representation.

from __future__ import (absolute_import, division, print_function, unicode_literals)

import six
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
colors_ = list(six.iteritems(colors.cnames))

# Add the single letter colors.
for name, rgb in six.iteritems(colors.ColorConverter.colors):
    hex_ = colors.rgb2hex(rgb)
    colors_.append((name, hex_))

# Transform to hex color values.
hex_ = [color[1] for color in colors_]

# Get the rgb equivalent.
rgb = [colors.hex2color(color) for color in hex_]

# Get the hsv equivalent.
hsv = [colors.rgb_to_hsv(color) for color in rgb]

# Split the hsv values to sort.
hue = [color[0] for color in hsv]
sat = [color[1] for color in hsv]
val = [color[2] for color in hsv]

# Sort by hue, saturation and value.
ind = np.lexsort((val, sat, hue))
sorted_colors = [colors_[i] for i in ind]

n = len(sorted_colors)
ncols = 4
nrows = int(np.ceil(1. * n / ncols))

fig = plt.figure()
ax = fig.add_subplot(111)

X, Y = fig.get_dpi() * fig.get_size_inches()

# row height
h = Y / (nrows + 1)

# col width
w = X / ncols

for i, (name, color) in enumerate(sorted_colors):
    col = i % ncols
    row = int(i / ncols)
    y = Y - (row * h) - h

    xi_line = w * (col + 0.05)
    xf_line = w * (col + 0.25)
    xi_text = w * (col + 0.3)

    ax.text(xi_text, y, name, fontsize=(h * 0.8),
            horizontalalignment='left',
            verticalalignment='center')

# Add extra black line a little bit thicker to make
# clear colors more visible.
ax.hlines(y, xi_line, xf_line, color='black', linewidth=(h * 0.7))
ax.hlines(y + h * 0.1, xi_line, xf_line, color=color, linewidth=(h * 0.6))

ax.set_xlim(0, X)
ax.set_ylim(0, Y)
ax.set_axis_off()

fig.subplots_adjust(left=0, right=1, top=1, bottom=0, hspace=0, wspace=0)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
78.1 event_handling example code: close_event.py

[source code]

```python
from __future__ import print_function
import matplotlib.pyplot as plt

def handle_close(evt):
    print('Closed Figure!')

fig = plt.figure()
fig.canvas.mpl_connect('close_event', handle_close)

plt.text(0.35, 0.5, 'Close Me!', dict(size=30))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.2 event_handling example code: data_browser.py

[source code]

```python
import numpy as np

class PointBrowser:
    ""
    Click on a point to select and highlight it -- the data that generated the point will be shown in the lower axes. Use the 'n' and 'p' keys to browse through the next and previous points
    ""
    def __init__(self):
        self.lastind = 0

        self.text = ax.text(0.05, 0.95, 'selected: none',
                           transform=ax.transAxes, va='top')
```

1791
self.selected, = ax.plot([xs[0]], [ys[0]], 'o', ms=12, alpha=0.4, color='yellow', visible=False)

def onpress(self, event):
    if self.lastind is None: return
    if event.key not in ('n', 'p'): return
    if event.key=='n': inc = 1
    else: inc = -1
    self.lastind += inc
    self.lastind = np.clip(self.lastind, 0, len(xs)-1)
    self.update()

def onpick(self, event):
    if event.artist!=line: return True

    N = len(event.ind)
    if not N: return True

    # the click locations
    x = event.mouseevent.xdata
    y = event.mouseevent.ydata

    distances = np.hypot(x-xs[event.ind], y-ys[event.ind])
    indmin = distances.argmin()
    dataind = event.ind[indmin]

    self.lastind = dataind
    self.update()

def update(self):
    if self.lastind is None: return

    dataind = self.lastind

    ax2.cla()
    ax2.plot(X[dataind])

    ax2.text(0.05, 0.9, 'mu=%1.3f
    sigma=%1.3f' %(xs[dataind], ys[dataind]),
    transform=ax2.transAxes, va='top')
    ax2.set_ylim(-0.5, 1.5)
    self.selected.set_visible(True)
    self.selected.set_data(xs[dataind], ys[dataind])

    self.text.set_text('selected: %d' %dataind)
    fig.canvas.draw()

if __name__ == '__main__':
    import matplotlib.pyplot as plt
```python
X = np.random.rand(100, 200)
x = np.mean(X, axis=1)
y = np.std(X, axis=1)

fig, (ax, ax2) = plt.subplots(2, 1)
ax.set_title('click on point to plot time series')
line, = ax.plot(x, y, 'o', picker=5)  # 5 points tolerance
browser = PointBrowser()
fig.canvas.mpl_connect('pick_event', browser.onpick)
fig.canvas.mpl_connect('key_press_event', browser.onpress)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 78.3 event_handling example code: figure_axes_enter_leave.py

[source code]

```python
"
""" Illustrate the figure and axes enter and leave events by changing the frame colors on enter and leave """
"
from __future__ import print_function
import matplotlib.pyplot as plt

def enter_axes(event):
    print('enter_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()

def leave_axes(event):
    print('leave_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()

def enter_figure(event):
    print('enter_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('red')
    event.canvas.draw()

def leave_figure(event):
    print('leave_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1, (ax, ax2) = plt.subplots(2, 1)
fig1.suptitle('mouse hover over figure or axes to trigger events')
```

78.3. event_handling example code: figure_axes_enter_leave.py 1793
fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)

fig2, (ax, ax2) = plt.subplots(2, 1)
fig2.suptitle('mouse hover over figure or axes to trigger events')
fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.4 event_handling example code: idle_and_timeout.py

[source code]

```python
from __future__ import print_function

"""
Demonstrate/test the idle and timeout API

This is only tested on gtk so far and is a prototype implementation
"""
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

t = np.arange(0.0, 2.0, 0.01)
y1 = np.sin(2*np.pi*t)
y2 = np.cos(2*np.pi*t)
line1, = ax.plot(y1)
line2, = ax.plot(y2)

N = 100

def on_idle(event):
    on_idle.count += 1
    print('idle', on_idle.count)
    line1.set_ydata(np.sin(2*np.pi*t*(N-on_idle.count)/float(N)))
    event.canvas.draw()
    # test boolean return removal
    if on_idle.count==N:
        return False
    return True

on_idle.cid = None
on_idle.count = 0
```

"""
fig.canvas.mpl_connect('idle_event', on_idle)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.5 event_handling example code: keypress_demo.py

[source code]

#!/usr/bin/env python

"""
Show how to connect to keypress events
"""
from __future__ import print_function
import sys
import numpy as np
import matplotlib.pyplot as plt

def press(event):
    print('press', event.key)
    sys.stdout.flush()
    if event.key== 'x':
        visible = xl.get_visible()
        xl.set_visible(not visible)
        fig.canvas.draw()

fig, ax = plt.subplots()
fig.canvas.mpl_connect('key_press_event', press)

ax.plot(np.random.rand(12), np.random.rand(12), 'go')
xl = ax.set_xlabel('easy come, easy go')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.6 event_handling example code: lasso_demo.py

[source code]

"""
Show how to use a lasso to select a set of points and get the indices of the selected points. A callback is used to change the color of the selected points
"""
This is currently a proof-of-concept implementation (though it is usable as is). There will be some refinement of the API.

```python
from matplotlib.widgets import Lasso
from matplotlib.colors import colorConverter
from matplotlib.collections import RegularPolyCollection
from matplotlib import path
import matplotlib.pyplot as plt
from numpy import nonzero
from numpy.random import rand

class Datum(object):
    colorin = colorConverter.to_rgba('red')
    colorout = colorConverter.to_rgba('blue')
    def __init__(self, x, y, include=False):
        self.x = x
        self.y = y
        if include: self.color = self.colorin
        else: self.color = self.colorout

class LassoManager(object):
    def __init__(self, ax, data):
        self.axes = ax
        self.canvas = ax.figure.canvas
        self.data = data
        self.Nxy = len(data)
        facecolors = [d.color for d in data]
        self.xys = [(d.x, d.y) for d in data]
        fig = ax.figure
        self.collection = RegularPolyCollection(
            fig.dpi, 6, sizes=(100,),
            facecolors=facecolors,
            offsets = self.xys,
            transOffset = ax.transData)
        ax.add_collection(self.collection)

        self.cid = self.canvas.mpl_connect('button_press_event', self.onpress)

    def callback(self, verts):
        facecolors = self.collection.get_facecolors()
        p = path.Path(verts)
        ind = p.contains_points(self.xys)
        for i in range(len(self.xys)):
            if ind[i]:
                facecolors[i] = Datum.colorin
            else:
                facecolors[i] = Datum.colorout
```

1796 Chapter 78. event_handling Examples
def on_press(self, event):
    if self.canvas.widgetlock.locked():
        return
    if event.inaxes is None:
        return
    self.lasso = Lasso(event.inaxes, (event.xdata, event.ydata), self.callback)
    # acquire a lock on the widget drawing
    self.canvas.widgetlock(self.lasso)

if __name__ == '__main__':
    data = [Datum(*xy) for xy in rand(100, 2)]
    ax = plt.axes(xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
    lman = LassoManager(ax, data)
    plt.show()
```python
def onpick(event):
    # on the pick event, find the orig line corresponding to the
    # legend proxy line, and toggle the visibility
    legline = event.artist
    origline = lined[legline]
    vis = not origline.get_visible()
    origline.set_visible(vis)
    # Change the alpha on the line in the legend so we can see what lines
    # have been toggled
    if vis:
        legline.set_alpha(1.0)
    else:
        legline.set_alpha(0.2)
    fig.canvas.draw()

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 78.8 event_handling example code: looking_glass.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
x, y = np.random.rand(2, 200)
fig, ax = plt.subplots()
circ = patches.Circle((0.5, 0.5), 0.25, alpha=0.8, fc='yellow')
ax.add_patch(circ)

ax.plot(x, y, alpha=0.2)
line, = ax.plot(x, y, alpha=1.0, clip_path=circ)

class EventHandler:
    def __init__(self):
        fig.canvas.mpl_connect('button_press_event', self.onpress)
        fig.canvas.mpl_connect('button_release_event', self.onrelease)
        fig.canvas.mpl_connect('motion_notify_event', self.onmove)
        self.x0, self.y0 = circ.center
        self.pressevent = None

    def onpress(self, event):
        if event.inaxes!=ax:
            return
```

1798 Chapter 78. event_handling Examples
if not circ.contains(event)[0]:
    return

self.pressevent = event

def onrelease(self, event):
    self.pressevent = None
    self.x0, self.y0 = circ.center

def onmove(self, event):
    if self.pressevent is None or event.inaxes!=self.pressevent.inaxes:
        return

dx = event.xdata - self.pressevent.xdata
dy = event.ydata - self.pressevent.ydata
circ.center = self.x0 + dx, self.y0 + dy
line.set_clip_path(circ)
fig.canvas.draw()

handler = EventHandler()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.9 event_handling example code: path_editor.py

[source code]

import numpy as np
import matplotlib.path as mpath
import matplotlib.patches as patches
import matplotlib.pyplot as plt

Path = mpath.Path

fig, ax = plt.subplots()

pathdata = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
]

codes, verts = zip(*pathdata)
path = mpath.Path(verts, codes)
patch = mpatches.PathPatch(path, facecolor="green", edgecolor="yellow", alpha=0.5)
ax.add_patch(patch)

class PathInteractor:
    """
    An path editor.
    
    Key-bindings
    
    't' toggle vertex markers on and off. When vertex markers are on, you can move them, delete them
    """

    showverts = True
    epsilon = 5  # max pixel distance to count as a vertex hit

def __init__(self, pathpatch):
    self.ax = pathpatch.axes
    canvas = self.ax.figure.canvas
    self.pathpatch = pathpatch
    self.pathpatch.set_animated(True)
    x, y = zip(*self.pathpatch.get_path().vertices)
    self.line, = ax.plot(x,y,marker='o', markerfacecolor='r', animated=True)
    self._ind = None  # the active vert
    canvas.mpl_connect('draw_event', self.draw_callback)
    canvas.mpl_connect('button_press_event', self.button_press_callback)
    canvas.mpl_connect('button_release_event', self.button_release_callback)
    canvas.mpl_connect('motion_notify_event', self.motion_notify_callback)
    self.canvas = canvas

    def draw_callback(self, event):
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)
        self.ax.draw_artist(self.pathpatch)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)

    def pathpatch_changed(self, pathpatch):
        """this method is called whenever the pathpatchgon object is called"
        # only copy the artist props to the line (except visibility)
        vis = self.line.get_visible()
        plt.Artist.update_from(self.line, pathpatch)
        self.line.set_visible(vis)  # don't use the pathpatch visibility state
def get_ind_under_point(self, event):
    'get the index of the vertex under point if within epsilon tolerance'

    # display coords
    xy = np.asarray(self.pathpatch.get_path().vertices)
    xyt = self.pathpatch.get_transform().transform(xy)
    xt, yt = xyt[:, 0], xyt[:, 1]
    d = np.sqrt((xt-event.x)**2 + (yt-event.y)**2)
    ind = d.argmin()

    if d[ind]>=self.epsilon:
        ind = None

    return ind

def button_press_callback(self, event):
    'whenever a mouse button is pressed'
    if not self.showverts: return
    if event.inaxes==None: return
    if event.button != 1: return
    self._ind = self.get_ind_under_point(event)

def button_release_callback(self, event):
    'whenever a mouse button is released'
    if not self.showverts: return
    if event.button != 1: return
    self._ind = None

def key_press_callback(self, event):
    'whenever a key is pressed'
    if not event.inaxes: return
    if event.key=='t':
        self.showverts = not self.showverts
        self.line.set_visible(self.showverts)
        if not self.showverts: self._ind = None

    self.canvas.draw()

def motion_notify_callback(self, event):
    'on mouse movement'
    if not self.showverts: return
    if self._ind is None: return
    if event.inaxes is None: return
    if event.button != 1: return
    x,y = event.xdata, event.ydata

    vertices = self.pathpatch.get_path().vertices

    vertices[self._ind] = x,y
    self.line.set_data(zip(*vertices))

    self.canvas.restore_region(self.background)
```python
self.ax.draw_artist(self.pathpatch)
self.ax.draw_artist(self.line)
self.canvas.blit(self.ax.bbox)
```

interactor = PathInteractor(patch)
ax.set_title('drag vertices to update path')
ax.set_xlim(-3, 4)
ax.set_ylim(-3, 4)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 78.10 event_handling example code: pick_event_demo.py

[source code]

```bash
#!/usr/bin/env python

"""
You can enable picking by setting the "picker" property of an artist
(for example, a matplotlib Line2D, Text, Patch, Polygon, AxesImage, etc...)

There are a variety of meanings of the picker property

None - picking is disabled for this artist (default)

boolean - if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist

float - if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, for example, the indices of the data within epsilon of the pick event

function - if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event.

    hit, props = picker(artist, mouseevent)

to determine the hit test. If the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes
```
After you have enabled an artist for picking by setting the "picker" property, you need to connect to the figure canvas pick_event to get pick callbacks on mouse press events. For example,

```python
def pick_handler(event):
    mouseevent = event.mouseevent
    artist = event.artist
    # now do something with this...
```

The pick event (matplotlib.backend_bases.PickEvent) which is passed to your callback is always fired with two attributes:

- **mouseevent** - the mouse event that generate the pick event. The mouse event in turn has attributes like x and y (the coordinates in display space, such as pixels from left, bottom) and xdata, ydata (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which Axes the mouse is over, etc. See matplotlib.backend_bases.MouseEvent for details.

- **artist** - the matplotlib.artist that generated the pick event.

Additionally, certain artists like Line2D and PatchCollection may attach additional meta data like the indices into the data that meet the picker criteria (for example, all the points in the line that are within the specified epsilon tolerance)

Examples below illustrate each of these methods.

```python
from __future__ import print_function
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.patches import Rectangle
from matplotlib.text import Text
from matplotlib.image import AxesImage
import numpy as np
import numpy.random

if 1:
    # simple picking, lines, rectangles and text
    fig, (ax1, ax2) = plt.subplots(2,1)
    ax1.set_title('click on points, rectangles or text', picker=True)
    ax1.set_ylabel('ylabel', picker=True, bbox=dict(facecolor='red'))
    line, = ax1.plot(np.random.rand(100), 'o', picker=5)  # 5 points tolerance
    # pick the rectangle
    bars = ax2.bar(range(10), np.random.rand(10), picker=True)
    for label in ax2.get_xticklabels():  # make the xtick labels pickable
        label.set_picker(True)
```

78.10. event_handling example code: pick_event_demo.py
def onpick1(event):
    if isinstance(event.artist, Line2D):
        thisline = event.artist
        xdata = thisline.get_xdata()
        ydata = thisline.get_ydata()
        ind = event.ind
        print('onpick1 line:', zip(np.take(xdata, ind), np.take(ydata, ind))
    elif isinstance(event.artist, Rectangle):
        patch = event.artist
        print('onpick1 patch:', patch.get_path())
    elif isinstance(event.artist, Text):
        text = event.artist
        print('onpick1 text:', text.get_text())

fig.canvas.mpl_connect('pick_event', onpick1)

if 1:
    # picking with a custom hit test function
    # you can define custom pickers by setting picker to a callable
    # function. The function has the signature
    # hit, props = func(artist, mouseevent)
    # to determine the hit test. if the mouse event is over the artist,
    # return hit=True and props is a dictionary of
    # properties you want added to the PickEvent attributes

def line_picker(line, mouseevent):
    """
    find the points within a certain distance from the mouseclick in
    data coords and attach some extra attributes, pickx and picky
    which are the data points that were picked
    """
    if mouseevent.xdata is None: return False, dict()
    xdata = line.get_xdata()
    ydata = line.get_ydata()
    maxd = 0.05
    d = np.sqrt((xdata-mouseevent.xdata)**2. + (ydata-mouseevent.ydata)**2.)
    ind = np.nonzero(np.less_equal(d, maxd))
    if len(ind):
        pickx = np.take(xdata, ind)
        picky = np.take(ydata, ind)
        props = dict(ind=ind, pickx=pickx, picky=picky)
        return True, props
    else:
        return False, dict()

def onpick2(event):
    print('onpick2 line:', event.pickx, event.picky)

fig, ax = plt.subplots()
ax.set_title('custom picker for line data')
line, = ax.plot(rand(100), rand(100), 'o', picker=line_picker)
fig.canvas.mpl_connect('pick_event', onpick2)

if 1: # picking on a scatter plot (matplotlib.collections.RegularPolyCollection)
x, y, c, s = rand(4, 100)
def onpick3(event):
    ind = event.ind
    print('onpick3 scatter:', ind, np.take(x, ind), np.take(y, ind))

fig, ax = plt.subplots()
col = ax.scatter(x, y, 100*s, c, picker=True)
#fig.savefig('pscoll.png')
fig.canvas.mpl_connect('pick_event', onpick3)

if 1: # picking images (matplotlib.image.AxesImage)
    fig, ax = plt.subplots()
    im1 = ax.imshow(rand(10,5), extent=(1,2,1,2), picker=True)
    im2 = ax.imshow(rand(5,10), extent=(3,4,1,2), picker=True)
    im3 = ax.imshow(rand(20,25), extent=(1,2,3,4), picker=True)
    im4 = ax.imshow(rand(30,12), extent=(3,4,3,4), picker=True)
    ax.axis([0,5,0,5])

def onpick4(event):
    artist = event.artist
    if isinstance(artist, AxesImage):
        im = artist
        A = im.get_array()
        print('onpick4 image', A.shape)

fig.canvas.mpl_connect('pick_event', onpick4)

plt.show()
X = numpy.random.rand(100, 1000)
xs = numpy.mean(X, axis=1)
ys = numpy.std(X, axis=1)

fig, ax = plt.subplots()
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance

def onpick(event):
    if event.artist!=line: return True
    N = len(event.ind)
    if not N: return True

    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N,1,subplotnum+1)
        ax.plot(X[dataind])
        ax.text(0.05, 0.9, 'mu=%1.3f
sigma=%1.3f
%(xs[dataind], ys[dataind]),
transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()
    return True

fig.canvas.mpl_connect('pick_event', onpick)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.12 event_handling example code: pipong.py

[source code]

#!/usr/bin/env python
# A matplotlib based game of Pong illustrating one way to write interactive
# animation which are easily ported to multiple backends
# pipong.py was written by Paul Ivanov <http://pirsquared.org>

from __future__ import print_function

import numpy as np
import matplotlib.pyplot as plt
from numpy.random import randn, randint

instructions = ""
Player A: Player B:
'e'  up  'i'
'd'  down  'k'

press 't' -- close these instructions
  (animation will be much faster)
press 'a' -- add a puck
press 'A' -- remove a puck
press 'l' -- slow down all pucks
press '2' -- speed up all pucks
press '3' -- slow down distractors
press '4' -- speed up distractors
press ' ' -- reset the first puck
press 'n' -- toggle distractors on/off
press 'g' -- toggle the game on/off

class Pad(object):
    def __init__(self, disp, x, y, type='l'):
        self.disp = disp
        self.x = x
        self.y = y
        self.w = 0.3
        self.score = 0
        self.xoffset = 0.3
        self.yoffset = 0.1
        if type=='r':
            self.xoffset *= -1.0
        if type=='l' or type=='r':
            self.signx = -1.0
            self.signy = 1.0
        else:
            self.signx = 1.0
            self.signy = -1.0
    def contains(self, loc):
        return self.disp.get_bbox().contains(loc.x, loc.y)

class Puck(object):
    def __init__(self, disp, pad, field):
        self.vmax = 0.2
        self.disp = disp
        self.field = field
        self._reset(pad)
    def _reset(self, pad):
        self.x = pad.x + pad.xoffset
        if pad.y < 0:
            self.y = pad.y + pad.yoffset
        else:
            self.y = pad.y - pad.yoffset
        self.vx = pad.x - self.x
        self.vy = pad.y + pad.w/2 - self.y
        self._speedlimit()
self._slower()
self._slower()
def update(self, pads):
    self.x += self.vx
    self.y += self.vy
    for pad in pads:
        if pad.contains(self):
            self.vx *= 1.2 * pad.signx
            self.vy *= 1.2 * pad.signy
    fudge = .001
    # probably cleaner with something like... if not self.field.contains(self.x, self.y):
    if self.x < 0 + fudge:
        #print "player A loses"
        pads[1].score += 1;
        self._reset(pads[0])
        return True
    if self.x > 7 - fudge:
        #print "player B loses"
        pads[0].score += 1;
        self._reset(pads[1])
        return True
    if self.y < -1 + fudge or self.y > 1 - fudge:
        self.vy *= -1.0
        # add some randomness, just to make it interesting
        self.vy = (randn() / 300.0 + 1 / 300.0) * np.sign(self.vy)
        self._speedlimit()
        return False
def _slower(self):
    self.vx /= 5.0
    self.vy /= 5.0
def _faster(self):
    self.vx *= 5.0
    self.vy *= 5.0
def _speedlimit(self):
    if self.vx > self.vmax:
        self.vx = self.vmax
    if self.vx < -self.vmax:
        self.vx = -self.vmax
    if self.vy > self.vmax:
        self.vy = self.vmax
    if self.vy < -self.vmax:
        self.vy = -self.vmax

class Game(object):
    def __init__(self, ax):
        # create the initial line
        self.ax = ax
        padAx = padBx = .50
        padAy = padBy = .30
        padBx += 6.3
        pA, = self.ax.barh(padAy, .2, height=.3, color='k', alpha=.5, edgecolor='b', lw=2, label="Player B")
        pB, = self.ax.barh(padBy, .2, height=.3, color='b', alpha=.5, edgecolor='k', lw=2, label="Player A")
pB, = self.ax.barh(padBy,.2, height=.3, left=padBx, color='k',alpha=.5, edgecolor='r',lw=2,label="Player A",animated=True)

# distractors
self.x = np.arange(0,2.22*np.pi,0.01)
self.line, = self.ax.plot(self.x, np.sin(self.x),"r", animated=True, lw=4)
self.line2, = self.ax.plot(self.x, np.cos(self.x),"g", animated=True, lw=4)
self.line3, = self.ax.plot(self.x, np.cos(self.x),"g", animated=True, lw=4)
self.line4, = self.ax.plot(self.x, np.cos(self.x),"r", animated=True, lw=4)
self.centerline,= self.ax.plot([3.5,3.5], [1,-1],"k",alpha=.5, animated=True, lw=8)
self.puckdisp = self.ax.scatter([1],[1],label='_nolegend_', s=200,c='g',alpha=.9,animated=True)

self.canvas = self.ax.figure.canvas
self.background = None
self.cnt = 0
self.distract = True
self.res = 100.0
self.on = False
self.inst = True
# show instructions from the beginning
self.background = None
self.pads = []
self.pads.append( Pad(pA,0,padAy))
self.pads.append( Pad(pB,padBx,padBy,'r'))
self.pucks =[]
self.i = self.ax.annotate(instructions,(.5,0.5),
                        name='monospace',
                        verticalalignment='center',
                        horizontalalignment='center',
                        multialignment='left',
                        textcoords='axes fraction',animated=True )
self.canvas.mpl_connect('key_press_event', self.key_press)

def draw(self, evt):
    draw_artist = self.ax.draw_artist
    if self.background is None:
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)

    # show the distractors
    if self.distract:
        self.line.set_ydata(np.sin(self.x+self.cnt/self.res))
        self.line2.set_ydata(np.cos(self.x-self.cnt/self.res))
        self.line3.set_ydata(np.tan(self.x+self.cnt/self.res))
        self.line4.set_ydata(np.tan(self.x-self.cnt/self.res))
        draw_artist(self.line)
        draw_artist(self.line2)
        draw_artist(self.line3)
        draw_artist(self.line4)

    # show the instructions - this is very slow
    if self.inst:
        self.ax.draw_artist(self.i)
# pucks and pads
if self.on:
    self.ax.draw_artist(self.centerline)
    for pad in self.pads:
        pad.disp.set_y(pad.y)
        pad.disp.set_x(pad.x)
        self.ax.draw_artist(pad.disp)

    for puck in self.pucks:
        if puck.update(self.pads):
            self.pads[0].disp.set_label(" + str(self.pads[0].score))
            self.pads[1].disp.set_label(" + str(self.pads[1].score))
            self.ax.legend(loc='center')
            self.leg = self.ax.get_legend()
            self.leg.draw_frame(False) # don't draw the legend border
            self.leg.get_frame().set_alpha(.2)
            plt.setp(self.leg.get_texts(), fontweight='bold', fontsize='xx-large')
            self.leg.get_frame().set_facecolor('0.2')
            self.background = None
            self.ax.figure.canvas.draw()
    return True

puck.disp.set_offsets([puck.x, puck.y])
self.ax.draw_artist(puck.disp)

# just redraw the axes rectangle
self.canvas.blit(self.ax.bbox)

if self.cnt==5000:
    # just so we don't get carried away
    print("...and you've been playing for too long!!!")
    plt.close()

    self.cnt += 1
    return True

def key_press(self, event):
    if event.key == '3':
        self.res *= 5.0
    if event.key == '4':
        self.res /= 5.0

    if event.key == 'e':
        self.pads[0].y += .1
        if self.pads[0].y > 1 - .3:
            self.pads[0].y = 1-.3
    if event.key == 'd':
        self.pads[0].y -= .1
        if self.pads[0].y < -1:
            self.pads[0].y = -1

    if event.key == 'i':
        #...
```
self.pads[1].y += .1
if self.pads[1].y > 1 - .3:
    self.pads[1].y = 1-.3
if event.key == 'k':
    self.pads[1].y -= .1
if self.pads[1].y < -1:
    self.pads[1].y = -1
if event.key == 'a':
    self.pucks.append(Puck(self.puckdisp, self.pads[randint(2)], self.ax.bbox))
if event.key == 'A' and len(self.pucks):
    self.pucks.pop()
if event.key == ' ' and len(self.pucks):
    self.pucks[0]._reset(self.pads[randint(2)])
if event.key == '1':
    for p in self.pucks:
        p._slower()
if event.key == '2':
    for p in self.pucks:
        p._faster()
if event.key == 'n':
    self.distract = not self.distract
if event.key == 'g':
    #self.ax.clear()
    self.on = not self.on
if event.key == 't':
    self.inst = not self.inst
    self.i.set_visible(self.i.get_visible())
if event.key == 'q':
    plt.close()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 78.13 event_handling example code: poly_editor.py

[source code]

```
""
This is an example to show how to build cross-GUI applications using matplotlib event handling to interact with objects on the canvas
""

import numpy as np
from matplotlib.lines import Line2D
from matplotlib.artist import Artist
from matplotlib.mlab import dist_point_to_segment

class PolygonInteractor:
```
An polygon editor.

Key-bindings

't' toggle vertex markers on and off. When vertex markers are on, you can move them, delete them.
'd' delete the vertex under point
'i' insert a vertex at point. You must be within epsilon of the line connecting two existing vertices.

```
showverts = True
epsilon = 5  # max pixel distance to count as a vertex hit

def __init__(self, ax, poly):
    if poly.figure is None:
        raise RuntimeError('You must first add the polygon to a figure or canvas before defining the interactor.
    self.ax = ax
    canvas = poly.figure.canvas
    self.poly = poly
    x, y = zip(*self.poly.xy)
    self.line = Line2D(x, y, marker='o', markerfacecolor='r', animated=True)
    self.ax.add_line(self.line)
    #self._update_line(poly)
    cid = self.poly.add_callback(self.poly_changed)
    self._ind = None  # the active vert
    canvas.mpl_connect('draw_event', self.draw_callback)
    canvas.mpl_connect('button_press_event', self.button_press_callback)
    canvas.mpl_connect('button_release_event', self.button_release_callback)
    canvas.mpl_connect('motion_notify_event', self.motion_notify_callback)
    self.canvas = canvas

    def draw_callback(self, event):
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)
        self.ax.draw_artist(self.poly)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)

    def poly_changed(self, poly):
        this method is called whenever the polygon object is called
        # only copy the artist props to the line (except visibility)
        vis = self.line.get_visible()
        Artist.update_from(self.line, poly)
        self.line.set_visible(vis)  # don't use the poly visibility state
```
def get_ind_under_point(self, event):
    # get the index of the vertex under point if within epsilon tolerance
    xy = np.asarray(self.poly.xy)
    xyt = self.poly.get_transform().transform(xy)
    xt, yt = xyt[:, :0], xyt[:, :1]
    d = np.sqrt((xt-event.x)**2 + (yt-event.y)**2)
    indseq = np.nonzero(np.equal(d, np.amin(d)))[0]
    ind = indseq[0]
    if d[ind] >= self.epsilon:
        ind = None
    return ind

def button_press_callback(self, event):
    # whenever a mouse button is pressed
    if not self.showverts: return
    if event.inaxes==None: return
    if event.button != 1: return
    self._ind = self.get_ind_under_point(event)

def button_release_callback(self, event):
    # whenever a mouse button is released
    if not self.showverts: return
    if event.button != 1: return
    self._ind = None

def key_press_callback(self, event):
    # whenever a key is pressed
    if not event.inaxes: return
    if event.key == 't':
        self.showverts = not self.showverts
        self.line.set_visible(self.showverts)
        if not self.showverts: self._ind = None
    elif event.key == 'd':
        ind = self.get_ind_under_point(event)
        if ind is not None:
            self.poly.xy = [tup for i, tup in enumerate(self.poly.xy) if i != ind]
            self.line.set_data(zip(*self.poly.xy))
    elif event.key == 'i':
        xys = self.poly.get_transform().transform(self.poly.xy)
        p = event.x, event.y
        for i in range(len(xys)-1):
            s0 = xys[i]
            s1 = xys[i+1]
            d = dist_point_to_segment(p, s0, s1)
            if d <= self.epsilon:
                self.poly.xy = np.array(
                    list(self.poly.xy[:i]) + 
                    [(event.xdata, event.ydata)] + 
                    list(self.poly.xy[i+1:]))
list(self.poly.xy[i:])

self.line.set_data(zip(*self.poly.xy))
break

self.canvas.draw()

def motion_notify_callback(self, event):
    'on mouse movement'
    if not self.showverts: return
    if self._ind is None: return
    if event.inaxes is None: return
    if event.button != 1: return
    x, y = event.xdata, event.ydata

    self.poly.xy[self._ind] = x, y
    self.line.set_data(zip(*self.poly.xy))

    self.canvas.restore_region(self.background)
    self.ax.draw_artist(self.poly)
    self.ax.draw_artist(self.line)
    self.canvas.blit(self.ax.bbox)

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    from matplotlib.patches import Polygon

    theta = np.arange(0, 2*np.pi, 0.1)
    r = 1.5

    xs = r*np.cos(theta)
    ys = r*np.sin(theta)

    poly = Polygon(list(zip(xs, ys)), animated=True)

    fig, ax = plt.subplots()
    ax.add_patch(poly)
    p = PolygonInteractor(ax, poly)

    #ax.add_line(p.line)
    ax.set_title('Click and drag a point to move it')
    ax.set_xlim((-2, 2))
    ax.set_ylim((-2, 2))
    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from __future__ import print_function

# For detailed comments on animation and the techniques used here, see
# the wiki entry
# http://www.scipy.org/wikis/topical_software/MatplotlibAnimation
import time

import gtk, gobject

import matplotlib
matplotlib.use('GTKAgg')

import matplotlib.pyplot as plt
import pipong

fig, ax = plt.subplots()
canvas = ax.figure.canvas

def start_anim(event):
    # gobject.idle_add(animation.draw,animation)
    gobject.timeout_add(10,animation.draw,animation)
    canvas.mpl_disconnect(start_anim.cid)

animation = pipong.Game(ax)
start_anim.cid = canvas.mpl_connect('draw_event', start_anim)

tstart = time.time()
plt.grid() # to ensure proper background restore
plt.show()
print('FPS:', animation.cnt/(time.time()-tstart))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
self.origXData = xdata
self.numpts = 3000
self.delta = xdata[-1] - xdata[0]

def resample(self, xstart, xend):
    # Very simple downsampling that takes the points within the range
    # and picks every Nth point
    mask = (self.origXData > xstart) & (self.origXData < xend)
    xdata = self.origXData[mask]
    ratio = int(xdata.size / self.numpts) + 1
    xdata = xdata[::ratio]
    ydata = self.origYData[mask]
    ydata = ydata[::ratio]

    return xdata, ydata

def update(self, ax):
    # Update the line
    lims = ax.viewLim
    if np.abs(lims.width - self.delta) > 1e-8:
        self.delta = lims.width
        xstart, xend = lims.intervalx
        self.line.set_data(*self.downsample(xstart, xend))
        ax.figure.canvas.draw_idle()

    # Read data
    data = wavread('/usr/share/sounds/purple/receive.wav')[0]
    ydata = np.tile(data[:, 0], 100)
    xdata = np.arange(ydata.size)

d = DataDisplayDownsampler(xdata, ydata)

fig, ax = plt.subplots()

# Hook up the line
xdata, ydata = d.downsample(xdata[0], xdata[-1])
d.line, = ax.plot(xdata, ydata)
d.set_autoscale_on(False)  # Otherwise, infinite loop

# Connect for changing the view limits
ax.callbacks.connect('xlim_changed', d.update)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.16 event_handling example code: test_mouseclicks.py

[source code]
#!/usr/bin/env python
from __future__ import print_function

import matplotlib
#matplotlib.use("WxAgg")
#matplotlib.use("TkAgg")
#matplotlib.use("GTKAgg")
#matplotlib.use("Qt4Agg")
#matplotlib.use("CocoaAgg")
#matplotlib.use("MacOSX")
import matplotlib.pyplot as plt

# print("***** TESTING WITH BACKEND: %s"%matplotlib.get_backend() + " *****")

def OnClick(event):
    if event.dblclick:
        print("DBLCLICK", event)
    else:
        print("DOWN", event)

def OnRelease(event):
    print("UP", event)

fig = plt.gcf()
cid_up = fig.canvas.mpl_connect('button_press_event', OnClick)
cid_down = fig.canvas.mpl_connect('button_release_event', OnRelease)

plt.gca().text(0.5, 0.5, "Click on the canvas to test mouse events.", ha="center", va="center")
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.17 event_handling example code: timers.py

[source code]

# Simple example of using general timer objects. This is used to update
# the time placed in the title of the figure.
import matplotlib.pyplot as plt
import numpy as np
from datetime import datetime

def update_title(axes):
    axes.set_title(datetime.now())
    axes.figure.canvas.draw()
fig, ax = plt.subplots()

x = np.linspace(-3, 3)
ax.plot(x, x*x)

# Create a new timer object. Set the interval to 100 milliseconds
# (1000 is default) and tell the timer what function should be called.
timer = fig.canvas.new_timer(interval=100)
timer.add_callback(update_title, ax)
timer.start()

# Or could start the timer on first figure draw
#def start_timer(evt):
#    timer.start()
#    fig.canvas.mpl_disconnect(drawid)
#drawid = fig.canvas.mpl_connect('draw_event', start_timer)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.18 event_handling example code: trifinder_event_demo.py

[source code]

"""
Example showing the use of a TriFinder object. As the mouse is moved over the
triangulation, the triangle under the cursor is highlighted and the index of
the triangle is displayed in the plot title.
"""

import matplotlib.pyplot as plt
from matplotlib.tri import Triangulation
from matplotlib.patches import Polygon
import numpy as np
import math

def update_polygon(tri):
    if tri == -1:
        points = [0, 0, 0]
    else:
        points = triangulation.triangles[tri]
x = triangulation.x[points]
y = triangulation.y[points]
polygon.set_xy(zip(x, y))

def motion_notify(event):
    if event.inaxes is None:
        tri = -1

"""
else:
    tri = trifinder(event.xdata, event.ydata)
    update_polygon(tri)
    plt.title('In triangle %i' % tri)
    event.canvas.draw()

# Create a Triangulation.
n_angles = 16
n_radii = 5
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi / n_angles
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
triangulation = Triangulation(x, y)
xmid = x[triangulation.triangles].mean(axis=1)
ymid = y[triangulation.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triangulation.set_mask(mask)

# Use the triangulation's default TriFinder object.
trifinder = triangulation.get_trifinder()

# Setup plot and callbacks.
plt.subplot(111, aspect='equal')
plt.triplot(triangulation, 'bo-')
polygon = Polygon([[0, 0], [0, 0]], facecolor='y')  # dummy data for xs, ys
update_polygon(-1)
plt.gca().add_patch(polygon)
plt.gcf().canvas.mpl_connect('motion_notify_event', motion_notify)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

78.19 event_handling example code: viewlims.py

[source code]

# Creates two identical panels. Zooming in on the right panel will show
# a rectangle in the first panel, denoting the zoomed region.
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

# We just subclass Rectangle so that it can be called with an Axes
# instance, causing the rectangle to update its shape to match the
# bounds of the Axes
class UpdatingRect(Rectangle):
```python
def __call__(self, ax):
    self.set_bounds(*ax.viewLim.bounds)
    ax.figure.canvas.draw_idle()

# A class that will regenerate a fractal set as we zoom in, so that you
# can actually see the increasing detail. A box in the left panel will show
# the area to which we are zoomed.
class MandlebrotDisplay(object):
    def __init__(self, h=500, w=500, niter=50, radius=2., power=2):
        self.height = h
        self.width = w
        self.niter = niter
        self.radius = radius
        self.power = power

def __call__(self, xstart, xend, ystart, yend):
    self.x = np.linspace(xstart, xend, self.width)
    self.y = np.linspace(ystart, yend, self.height).reshape(-1,1)
    c = self.x + 1.0j * self.y
    threshold_time = np.zeros((self.height, self.width))
    z = np.zeros(threshold_time.shape, dtype=np.complex)
    mask = np.ones(threshold_time.shape, dtype=np.bool)
    for i in range(self.niter):
        z[mask] = z[mask]**self.power + c[mask]
        mask = (np.abs(z) < self.radius)
        threshold_time += mask
    return threshold_time

    def ax_update(self, ax):
        ax.set_autoscale_on(False) # Otherwise, infinite loop

        # Get the number of points from the number of pixels in the window
        dims = ax.axesPatch.get_window_extent().bounds
        self.width = int(dims[2] + 0.5)
        self.height = int(dims[2] + 0.5)

        # Get the range for the new area
        xstart, ystart, xdelta, ydelta = ax.viewLim.bounds
        xend = xstart + xdelta
        yend = ystart + ydelta

        # Update the image object with our new data and extent
        im = ax.images[-1]
        im.set_data(self.__call__(xstart, xend, ystart, yend))
        im.set_extent((xstart, xend, ystart, yend))
        ax.figure.canvas.draw_idle()

md = MandlebrotDisplay()
Z = md(-2., 0.5, -1.25, 1.25)

fig1, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(Z, origin='lower', extent=(md.x.min(), md.x.max(), md.y.min(), md.y.max()))
ax2.imshow(Z, origin='lower', extent=(md.x.min(), md.x.max(), md.y.min(), md.y.max()))
```

1820 Chapter 78. event_handling Examples
rect = UpdatingRect([0, 0], 0, 0, facecolor='None', edgecolor='black')
rect.set_bounds(*ax2.viewLim.bounds)
ax1.add_patch(rect)

# Connect for changing the view limits
ax2.callbacks.connect('xlim_changed', rect)
ax2.callbacks.connect('ylim_changed', rect)

ax2.callbacks.connect('xlim_changed', md.ax_update)
ax2.callbacks.connect('ylim_changed', md.ax_update)

plt.show()
axzoom.set_xlim(x-0.1, x+0.1)
axzoom.set_ylim(y-0.1, y+0.1)
figzoom.canvas.draw()

figsrc.canvas.mpl_connect('button_press_event', onpress)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
CHAPTER

SEVENTYNINE

IMAGES_CONTOURS_AND_FIELDS EXAMPLES

79.1 images_contours_and_fields example code: image_demo.py

"""
Simple demo of the imshow function.
"""
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
image_file = cbook.get_sample_data('ada.png')
image = plt.imread(image_file)

1823


```python
plt.imshow(image)
plt.axis('off') # clear x- and y-axes
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 79.2 images_contours_and_fields example code: image_demo_clip_path.py

```python

Demo of image that's been clipped by a circular path.

```

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.cbook as cbook

image_file = cbook.get_sample_data('grace_hopper.png')
image = plt.imread(image_file)

fig, ax = plt.subplots()
```
im = ax.imshow(image)
patch = patches.Circle((260, 200), radius=200, transform=ax.transData)
im.set_clip_path(patch)

plt.axis('off')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

79.3 images_contours_and_fields example code: interpolation_methods.py

Show all different interpolation methods for imshow

import matplotlib.pyplot as plt
import numpy as np

# from the docs:
# If interpolation is None, default to rc image.interpolation. See also
# the filternorm and filterrad parameters. If interpolation is 'none', then
# no interpolation is performed on the Agg, ps and pdf backends. Other
# backends will fall back to 'nearest'.
#
# http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow

methods = [None, 'none', 'nearest', 'bilinear', 'bicubic', 'spline16',
          'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric',
          'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos']

""
Show all different interpolation methods for imshow
""
grid = np.random.rand(4, 4)

fig, axes = plt.subplots(3, 6, figsize=(12, 6), subplot_kw={'xticks': [], 'yticks': []})
fig.subplots_adjust(hspace=0.3, wspace=0.05)

for ax, interp_method in zip(axes.flat, methods):
    ax.imshow(grid, interpolation=interp_method)
    ax.set_title(interp_method)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
79.4 images_contours_and_fields example code: interpolation_none_vs_nearest.py

Saved as a PNG
Displays the difference between interpolation = 'none' and interpolation = 'nearest'.

Interpolation = 'none' and interpolation = 'nearest' are equivalent when converting a figure to an image file, such as a PNG. Interpolation = 'none' and interpolation = 'nearest' behave quite differently, however, when converting a figure to a vector graphics file, such as a PDF. As shown, Interpolation = 'none' works well when a big image is scaled down, while interpolation = 'nearest' works well when a small image is blown up.

```python
import numpy as np
import matplotlib.pyplot as plt
```
import matplotlib.cbook as cbook

# Load big image
big_im_path = cbook.get_sample_data('necked_tensile_specimen.png')
big_im = plt.imread(big_im_path)

# Define small image
small_im = np.array([[0.25, 0.75, 1.0, 0.75], [0.1, 0.65, 0.5, 0.4],
                     [0.6, 0.3, 0.0, 0.2], [0.7, 0.9, 0.4, 0.6]])

# Create a 2x2 table of plots
fig = plt.figure(figsize=[8.0, 7.5])
ax = plt.subplot(2,2,1)
ax.imshow(big_im, interpolation='none')
ax = plt.subplot(2,2,2)
ax.imshow(big_im, interpolation='nearest')
ax = plt.subplot(2,2,3)
ax.imshow(small_im, interpolation='none')
ax = plt.subplot(2,2,4)
ax.imshow(small_im, interpolation='nearest')
plt.subplots_adjust(left=0.24, wspace=0.2, hspace=0.1, 
                    bottom=0.05, top=0.86)

# Label the rows and columns of the table
fig.text(0.03, 0.645, 'Big Image
Scaled Down', ha='left')
fig.text(0.03, 0.225, 'Small Image
Blown Up', ha='left')
fig.text(0.383, 0.9, "Interpolation = 'none'", ha='center')
fig.text(0.75, 0.9, "Interpolation = 'nearest'", ha='center')

# If you were going to run this example on your local machine, you
# would save the figure as a PNG, save the same figure as a PDF, and
# then compare them. The following code would suffice.
txt = fig.text(0.452, 0.95, 'Saved as a PNG', fontsize=18)
# plt.savefig('None_vs_nearest-png.png')
# txt.set_text('Saved as a PDF')
# plt.savefig('None_vs_nearest-pdf.pdf')

# Here, however, we need to display the PDF on a webpage, which means
# the PDF must be converted into an image. For the purposes of this
# example, the 'Nearest_vs_none-pdf.pdf' has been pre-converted into
# 'Nearest_vs_none-pdf.png' at 80 dpi. We simply need to load and
# display it.
pdf_im_path = cbook.get_sample_data('Nearest_vs_none-pdf.png')
pdf_im = plt.imread(pdf_im_path)
fig2 = plt.figure(figsize=[8.0, 7.5])
plt.figimage(pdf_im)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
## images_contours_and_fields example code: pcolormesh_levels.py

```python
import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
import numpy as np

# make these smaller to increase the resolution
dx, dy = 0.05, 0.05

# generate 2d grids for the x & y bounds
y, x = np.mgrid[slice(1, 5 + dy, dy),
                slice(1, 5 + dx, dx)]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

---

Shows how to combine Normalization and Colormap instances to draw "levels" in pcolor, pcolormesh and imshow type plots in a similar way to the levels keyword argument to contour/contourf.

```python
import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
import numpy as np

# make these smaller to increase the resolution
dx, dy = 0.05, 0.05

# generate 2d grids for the x & y bounds
y, x = np.mgrid[slice(1, 5 + dy, dy),
                slice(1, 5 + dx, dx)]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```
# x and y are bounds, so z should be the value *inside* those bounds.
# Therefore, remove the last value from the z array.
z = z[:-1, :-1]
levels = MaxNLocator(nbins=15).tick_values(z.min(), z.max())

# pick the desired colormap, sensible levels, and define a normalization
# instance which takes data values and translates those into levels.
cmap = plt.get_cmap('PiYG')
norm = BoundaryNorm(levels, ncolors=cmap.N, clip=True)

plt.subplot(2, 1, 1)
im = plt.pcolormesh(x, y, z, cmap=cmap, norm=norm)
plt.colorbar()
# set the limits of the plot to the limits of the data
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.title('pcolormesh with levels')

plt.subplot(2, 1, 2)
# contours are *point* based plots, so convert our bound into point
# centers
plt.contourf(x[:-1, :-1] + dx / 2.,
y[:-1, :-1] + dy / 2., z, levels=levels,
cmap=cmap)
plt.colorbar()
plt.title('contourf with levels')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
79.6 images_contours_and_fields example code: streamplot_demo_features.py
Demo of the `streamplot` function.

A streamplot, or streamline plot, is used to display 2D vector fields. This example shows a few features of the stream plot function:

- Varying the color along a streamline.
- Varying the density of streamlines.
- Varying the line width along a streamline.

```python
import numpy as np
import matplotlib.pyplot as plt

Y, X = np.mgrid[-3:3:100j, -3:3:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
speed = np.sqrt(U*U + V*V)

plt.streamplot(X, Y, U, V, color=U, linewidth=2, cmap=plt.cm.autumn)
plt.colorbar()

f, (ax1, ax2) = plt.subplots(ncols=2)
ax1.streamplot(X, Y, U, V, density=[0.5, 1])
lw = 5*speed/speed.max()
```
ax2.streamplot(X, Y, U, V, density=0.6, color='k', linewidth=lw)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

79.7 images_contours_and_fields example code: streamplot_demo_masking.py

********
Demo of the streamplot function with masking.

This example shows how streamlines created by the streamplot function skips masked regions and NaN values.

********
import numpy as np
import matplotlib.pyplot as plt

w = 3
Y, X = np.mgrid[-w:w:100j, -w:w:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
speed = np.sqrt(U*U + V*V)

mask = np.zeros(U.shape, dtype=bool)
mask[40:60, 40:60] = 1
U = np.ma.array(U, mask=mask)
U[:20, :20] = np.nan

plt.streamplot(X, Y, U, V, color='r')

plt.imshow(~mask, extent=(-w, w, -w, w), alpha=0.5,
            interpolation='nearest', cmap=plt.cm.gray)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
80.1 lines_bars_and_markers example code: barh_demo.py

```python
import matplotlib.pyplot as plt
plt.rcdefaults()
import numpy as np

How fast do you want to go today?

Tom
Dick
Harry
Slim
Jim

Simple demo of a horizontal bar chart.
```
# Example data
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

plt.barh(y_pos, performance, xerr=error, align='center', alpha=0.4)
plt.yticks(y_pos, people)
plt.xlabel('Performance')
plt.title('How fast do you want to go today?')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

80.2 lines_bars_and_markers example code: fill_demo.py

""
Simple demo of the fill function.
""
import numpy as np
import matplotlib.pyplot as plt
```python
x = np.linspace(0, 1)
y = np.sin(4 * np.pi * x) * np.exp(-5 * x)

plt.fill(x, y, 'r')
plt.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

80.3 lines_bars_and_markers example code: fill_demo_features.py

""
Demo of the fill function with a few features.

In addition to the basic fill plot, this demo shows a few optional features:

* Multiple curves with a single command.
* Setting the fill color.
* Setting the opacity (alpha value).
""

import numpy as np
```python
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.sin(3 * x)
plt.fill(x, y1, 'b', x, y2, 'r', alpha=0.3)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

80.4 lines_bars_and_markers example code: line_demo_dash_control.py

""
Demo of a simple plot with a custom dashed line.

A Line object's `set_dashes` method allows you to specify dashes with
a series of on/off lengths (in points).
""
```python
import numpy as np
import matplotlib.pyplot as plt
```
```python
x = np.linspace(0, 10)
line, = plt.plot(x, np.sin(x), '--', linewidth=2)

dashes = [10, 5, 100, 5]  # 10 points on, 5 off, 100 on, 5 off
line.set_dashes(dashes)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

80.5 lines_bars_and_markers example code: line_styles_reference.py

line styles

```
```
```
```
```
```
```
```
```

""
Reference for line-styles included with Matplotlib.
""
import numpy as np
import matplotlib.pyplot as plt

color = 'cornflowerblue'
points = np.ones(5)  # Draw 5 points for each line
text_style = dict(horizontalalignment='right', verticalalignment='center',
```
def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()

def nice_repr(text):
    return repr(text).lstrip('u')

# Plot all line styles.
f, ax = plt.subplots()
linestyles = ['-','--', '-.', ':']
for y, linestyle in enumerate(linestyles):
    ax.text(-0.5, y, nice_repr(linestyle), **text_style)
    ax.plot(y * points, linestyle=linestyle, color=color, linewidth=3)
    format_axes(ax)
ax.set_title('line styles')

plt.show()
fill style

'none'

'top'

'bottom'

'right'

'left'

'full'

"""
Reference for marker fill-styles included with Matplotlib.
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

points = np.ones(5)  # Draw 3 points for each line
text_style = dict(horizontalalignment='right', verticalalignment='center',
                   fontsize=12, fontdict={'family': 'monospace'})
marker_style = dict(color='cornflowerblue', linestyle=':', marker='o',
                    markersize=15, markerfacecoloralt='gray')

def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()

def nice_repr(text):
    return repr(text).lstrip('u')
fig, ax = plt.subplots()

# Plot all fill styles.
for y, fill_style in enumerate(Line2D.fillStyles):
    ax.text(-0.5, y, nice_repr(fill_style), **text_style)
    ax.plot(y * points, fillstyle=fill_style, **marker_style)
    format_axes(ax)
    ax.set_title('fill style')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

80.7 lines_bars_and_markers example code: marker_reference.py

un-filled markers

', ' 0
',. ' 1
'1' 2
'2' 3
'3' 4
'4' 5
' ' 6
'x' 7
'|' ' +

un-filled markers
filled markers

'8'
'o'
'^'
'<'
'>'
'8'
's'
'p'
'*'
'h'
'H'
'D'
'd'

Reference for filled- and unfilled-marker types included with Matplotlib.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

points = np.ones(3)  # Draw 3 points for each line

marker_style = {'linestyle': ':', 'color': 'cornflowerblue', 'markersize': 10}

def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()

def nice_repr(text):
    return repr(text).lstrip('u')

def split_list(a_list):
    i_half = len(a_list) // 2

80.7. lines_bars_and_markers example code: marker_reference.py
```
return (a_list[:i_half], a_list[i_half:])

# Plot all un-filled markers
# --------------------------

fig, axes = plt.subplots(ncols=2)

# Filter out filled markers and marker settings that do nothing.
unfilled_markers = [m for m, func in Line2D.markers.iteritems()
    if func != 'nothing' and m not in Line2D.filled_markers]
unfilled_markers = sorted(unfilled_markers)[::-1]  # Reverse-sort for pretty
for ax, markers in zip(axes, split_list(unfilled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, nice_repr(marker), **text_style)
        ax.plot(y * points, marker=marker, **marker_style)
        format_axes(ax)
fig.suptitle('un-filled markers', fontsize=14)

# Plot all filled markers.
# ------------------------

fig, axes = plt.subplots(ncols=2)
for ax, markers in zip(axes, split_list(Line2D.filled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, nice_repr(marker), **text_style)
        ax.plot(y * points, marker=marker, **marker_style)
        format_axes(ax)
fig.suptitle('filled markers', fontsize=14)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
from numpy.random import rand

for color in ['red', 'green', 'blue']:
    n = 750
    x, y = rand(2, n)
    scale = 200.0 * rand(n)
    plt.scatter(x, y, c=color, s=scale, label=color,
                alpha=0.3, edgecolors='none')

plt.legend()
plt.grid(True)
plt.show()
81.1 misc example code: contour_manual.py

```python
import matplotlib.pyplot as plt
from matplotlib.contour import ContourSet
import matplotlib.cm as cm

# Contour lines for each level are a list/tuple of polygons.
lines0 = [[[0,0],[0,4]]]
lines1 = [[[2,0],[1,2],[1,3]]]
lines2 = [[[3,0],[3,2]], [[3,3],[3,4]]]  # Note two lines.

# Filled contours between two levels are also a list/tuple of polygons.
# Points can be ordered clockwise or anticlockwise.
filled01 = [[[0,0],[0,4],[1,3],[1,2],[2,0]]]
filled12 = [[[2,0],[3,0],[3,2],[1,3],[1,2]],  # Note two polygons.
             [[1,4],[3,4],[3,3]]]

plt.figure()

# Filled contours using filled=True.
cs = ContourSet(plt.gca(), [0,1,2], [filled01, filled12], filled=True, cmap=cm.bone)
cbar = plt.colorbar(cs)

# Contour lines (non-filled).
lines = ContourSet(plt.gca(), [0,1,2], [lines0, lines1, lines2], cmap=cm.cool,
                   linewidths=3)
cbar.add_lines(lines)

plt.axis([-0.5, 3.5, -0.5, 4.5])
plt.title('User-specified contours')
```
# Multiple filled contour lines can be specified in a single list of polygon
# vertices along with a list of vertex kinds (code types) as described in the
# Path class. This is particularly useful for polygons with holes.
# Here a code type of 1 is a MOVETO, and 2 is a LINETO.

plt.figure()
filled01 = [[0,0],[3,0],[3,3],[0,3],[1,1],[1,2],[2,2],[2,1]]
kinds01 = [[1,2,2,2,1,2,2,2]]

cs = ContourSet(plt.gca(), [0,1], [filled01], [kinds01], filled=True)
cbar = plt.colorbar(cs)

plt.axis([-0.5, 3.5, -0.5, 3.5])
plt.title('User specified filled contours with holes')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.2 misc example code: font_indexing.py

[source code]

""
A little example that shows how the various indexing into the font
tables relate to one another. Mainly for mpl developers....
""

from __future__ import print_function
import matplotlib
from matplotlib.ft2font import FT2Font, KERNING_DEFAULT, KERNING_UNFITTED, KERNING_UNScaled

#fname = '/usr/share/fonts/sfd/FreeSans.ttf'
fname = matplotlib.get_data_path() + '/fonts/ttf/Vera.ttf'
font = FT2Font(fname)
font.set_charmap(0)

codes = font.get_charmap().items()
#dsu = [(ccode, glyphind) for ccode, glyphind in codes]
#dsu.sort()
#for ccode, glyphind in dsu:
#    try: name = font.get_glyph_name(glyphind)
#    except RuntimeError: pass
#    else: print % 4d % 4d %s %s%(glyphind, ccode, hex(int(ccode)), name)

# make a charname to charcode and glyphind dictionary
coded = {}
glyphd = {}

1850  Chapter 81. misc Examples
for ccode, glyphind in codes:
    name = font.get_glyph_name(glyphind)
    coded[name] = ccode
    glyphd[name] = glyphind

code = coded['A']
glyph = font.load_char(code)

# print glyph.bbox
print(glyphd['A'], glyphd['V'], coded['A'], coded['V'])
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_DEFAULT))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNFITTED))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNSCALED))
print('AV', font.get_kerning(glyphd['A'], glyphd['T'], KERNING_UNSCALED))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.3 misc example code: ftface_props.py

[source code]

#!/usr/bin/env python

from __future__ import print_function

""
This is a demo script to show you how to use all the properties of an
FT2Font object. These describe global font properties. For
individual character metrics, use the Glyp object, as returned by
load_char
""

import matplotlib
from matplotlib.ft2font import FT2Font

#fname = '/usr/local/share/matplotlib/VeraIt.ttf'
fname = matplotlib.get_data_path() + '/fonts/ttf/VeraIt.ttf'
#fname = '/usr/local/share/matplotlib/cmr10.ttf'

font = FT2Font(fname)

# these constants are used to access the style_flags and face_flags
FT_FACE_FLAG_SCALABLE = 1 << 0
FT_FACE_FLAG_FIXED_SIZES = 1 << 1
FT_FACE_FLAG_FIXED_WIDTH = 1 << 2
FT_FACE_FLAG_SFNT = 1 << 3
FT_FACE_FLAG_HORIZONTAL = 1 << 4
FT_FACE_FLAG_VERTICAL = 1 << 5
FT_FACE_FLAG_KERNING = 1 << 6
FT_FACE_FLAG_FAST_GLYPHS = 1 << 7
FT_FACE_FLAG_MULTIPLE_MASTERS = 1 << 8
FT_FACE_FLAG_GLYPH_NAMES = 1 << 9
FT_FACE_FLAG_EXTERNAL_STREAM = 1 << 10
FT_STYLE_FLAG_ITALIC = 1 << 0
FT_STYLE_FLAG_BOLD = 1 << 1

print('Num faces :', font.num_faces)  # number of faces in file
print('Num glyphs :', font.num_glyphs)  # number of glyphs in the face
print('Family name :', font.family_name)  # face family name
print('Style name :', font.style_name)  # face style name
print('PS name :', font.postscript_name)  # the postscript name
print('Num fixed :', font.num_fixed_sizes)  # number of embedded bitmap in face

# the following are only available if face.scalable
if font.scalable:
    # the face global bounding box (xmin, ymin, xmax, ymax)
    print('Bbox :', font.bbox)
    # number of font units covered by the EM
    print('EM :', font.units_per_EM)
    # the ascender in 26.6 units
    print('Ascender :', font.ascender)
    # the descender in 26.6 units
    print('Descender :', font.descender)
    # the height in 26.6 units
    print('Height :', font.height)
    # maximum horizontal cursor advance
    print('Max adv width :', font.max_advance_width)
    # same for vertical layout
    print('Max adv height :', font.max_advance_height)
    # vertical position of the underline bar
    print('Underline pos :', font.underline_position)
    # vertical thickness of the underline
    print('Underline thickness :', font.underline_thickness)

print('Italics :', font.style_flags & FT_STYLE_FLAG_ITALIC != 0)
print('Bold :', font.style_flags & FT_STYLE_FLAG_BOLD != 0)
print('Scalable :', font.style_flags & FT_FACE_FLAG_SCALABLE != 0)
print('Fixed sizes :', font.style_flags & FT_FACE_FLAG_FIXED_SIZES != 0)
print('Fixed width :', font.style_flags & FT_FACE_FLAG_FIXED_WIDTH != 0)
print('SFNT :', font.style_flags & FT_FACE_FLAG_SFNT != 0)
print('Horizontal :', font.style_flags & FT_FACE_FLAG_HORIZONTAL != 0)
print('Vertical :', font.style_flags & FT_FACE_FLAG_VERTICAL != 0)
print('Kerning :', font.style_flags & FT_FACE_FLAG_KERNING != 0)
print('Fast glyphs :', font.style_flags & FT_FACE_FLAG_FAST_GLYPHS != 0)
print('Mult. masters :', font.style_flags & FT_FACE_FLAG_MULTIPLE_MASTERS != 0)
print('Glyph names :', font.style_flags & FT_FACE_FLAG_GLYPH_NAMES != 0)

print(dir(font))
cmap = font.get_charmap()
print(font.get_kerning)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
81.4 misc example code: image_thumbnail.py

[source code]

"""
You can use matplotlib to generate thumbnails from existing images.
matplotlib natively supports PNG files on the input side, and other
image types transparently if you have PIL installed
"""

from __future__ import print_function
# build thumbnails of all images in a directory
import sys, os, glob
import matplotlib.image as image

if len(sys.argv)!=2:
    print('Usage: python %s IMAGEDIR' % __file__)
    raise SystemExit
indir = sys.argv[1]
if not os.path.isdir(indir):
    print('Could not find input directory "%s"' % indir)
    raise SystemExit

outdir = 'thumbs'
if not os.path.exists(outdir):
    os.makedirs(outdir)

for fname in glob.glob(os.path.join(indir, '*.png')):
basedir, basename = os.path.split(fname)
outfile = os.path.join(outdir, basename)
fig = image.thumbnail(fname, outfile, scale=0.15)
print('saved thumbnail of %s to %s' % (fname, outfile))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.5 misc example code: longshort.py

[source code]

"""
Illustrate the rec array utility funcitons by loading prices from a
csv file, computing the daily returns, appending the results to the
record arrays, joining on date
"""

import urllib
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
# grab the price data off yahoo


# load the CSV files into record arrays
r1 = mlab.csv2rec(file(u1[0]))
r2 = mlab.csv2rec(file(u2[0]))

# compute the daily returns and add these columns to the arrays
gains1 = np.zeros_like(r1.adj_close)
gains2 = np.zeros_like(r2.adj_close)
gains1[1:] = np.diff(r1.adj_close)/r1.adj_close[:-1]
gains2[1:] = np.diff(r2.adj_close)/r2.adj_close[:-1]

r1 = mlab.rec_append_fields(r1, 'gains', gains1)
r2 = mlab.rec_append_fields(r2, 'gains', gains2)

# now join them by date; the default postfixes are 1 and 2. The
# default jointype is inner so it will do an intersection of dates and
# drop the dates in AAPL which occurred before GOOG started trading in
# 2004. r1 and r2 are reverse ordered by date since Yahoo returns
# most recent first in the CSV files, but rec_join will sort by key so
# r below will be properly sorted
r = mlab.rec_join('date', r1, r2)

g = r.gains1 - r.gains2
tr = (1+g).cumprod()  # the total return

# plot the return
fig, ax = plt.subplots()
ax.plot(r.date, tr)
ax.set_title('total return: long APPL, short GOOG')
ax.grid()
fig.autofmt_xdate()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.6 misc example code: multiprocess.py

[source code]

#Demo of using multiprocessing for generating data in one process and plotting
#in another.
#Written by Robert Cimrman
#Requires >= Python 2.6 for the multiprocessing module or having the
#standalone processing module installed

from __future__ import print_function
import time
try:
    from multiprocessing import Process, Pipe
except ImportError:
    from processing import Process, Pipe
import numpy as np

import matplotlib
matplotlib.use('GtkAgg')
import matplotlib.pyplot as plt
import gobject

class ProcessPlotter(object):
    def __init__(self):
        self.x = []
        self.y = []

    def terminate(self):
        plt.close('all')

    def poll_draw(self):
        def call_back():
            while 1:
                if not self.pipe.poll():
                    break

                command = self.pipe.recv()

                if command is None:
                    self.terminate()
                    return False
                else:
                    self.x.append(command[0])
                    self.y.append(command[1])
                    self.ax.plot(self.x, self.y, 'ro')

                    self.fig.canvas.draw()
                    return True

        return call_back

    def __call__(self, pipe):
        print('starting plotter...')

        self.pipe = pipe
        self.fig, self.ax = plt.subplots()
        self.gid = gobject.timeout_add(1000, self.poll_draw())

        print('...done')
        plt.show()
class NBPlot(object):
    def __init__(self):
        self.plot_pipe, plotter_pipe = Pipe()
        self.plotter = ProcessPlotter()
        self.plot_process = Process(target = self.plotter,
                                     args = (plotter_pipe,))
        self.plot_process.daemon = True
        self.plot_process.start()

def plot(self, finished=False):
    send = self.plot_pipe.send
    if finished:
        send(None)
    else:
        data = np.random.random(2)
        send(data)

def main():
    pl = NBPlot()
    for ii in range(10):
        pl.plot()
        time.sleep(0.5)
        raw_input('press Enter...')
    pl.plot(finished=True)

if __name__ == '__main__':
    main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.7 misc example code: rasterization_demo.py

[source code]

import numpy as np
import matplotlib.pyplot as plt

d = np.arange(100).reshape(10, 10)
x, y = np.meshgrid(np.arange(11), np.arange(11))
theta = 0.25*np.pi
xx = x*np.cos(theta) - y*np.sin(theta)
yy = x*np.sin(theta) + y*np.cos(theta)

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
ax1.pcolormesh(xx, yy, d)
ax1.set_title('No Rasterization')
ax2.pcolormesh(xx, yy, d)
ax2.set_title('Rasterization')
m = ax2.pcolormesh(xx, yy, d)
m.set_rasterized(True)

ax3.set_aspect(1)
ax3.pcolormesh(xx, yy, d)
ax3.text(0.5, 0.5, "Text", alpha=0.2,
        va="center", ha="center", size=50, transform=ax3.transAxes)

ax3.set_title("No Rasterization")

ax4.set_aspect(1)
m = ax4.pcolormesh(xx, yy, d)
m.set_zorder(-20)

ax4.text(0.5, 0.5, "Text", alpha=0.2,
        zorder=-15,
        va="center", ha="center", size=50, transform=ax4.transAxes)

ax4.set_rasterization_zorder(-10)

ax4.set_title("Rasterization z$<-10\$")

# ax2.title.set_rasterized(True)  # should display a warning

plt.savefig("test_rasterization.pdf", dpi=150)
plt.savefig("test_rasterization.eps", dpi=150)

if not plt.rcParams["text.usetex"]:
    plt.savefig("test_rasterization.svg", dpi=150)
    # svg backend currently ignores the dpi

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.8 misc example code: rc_traits.py

[source code]

# Here is some example code showing how to define some representative
# rc properties and construct a matplotlib artist using traits.
# matplotlib does not ship with enthought.traits, so you will need to
# install it separately.

from __future__ import print_function

import sys, os, re
import traits.api as traits
from matplotlib.cbook import is_string_like
from matplotlib.artist import Artist
```python
doprint = True
flexible_true_trait = traits.Trait(
    True,
    { 'true': True, 't': True, 'yes': True, 'y': True, 'on': True, True: True,
      'false': False, 'f': False, 'no': False, 'n': False, 'off': False, False: False
    }
)
flexible_false_trait = traits.Trait( False, flexible_true_trait )

colors = {
    'c': '#00bfbf',
    'b': '#0000ff',
    'g': '#008000',
    'k': '#000000',
    'm': '#bf00bf',
    'r': '#ff0000',
    'w': '#ffffff',
    'y': '#bfbf00',
    'gold': '#FFD700',
    'peachpuff': '#FFDAB9',
    'navajowhite': '#FFDEAD',
}

def hex2color(s):
    """Convert hex string (like html uses, e.g., #efefef) to a r,g,b tuple""
    return tuple([int(n, 16)/255.0 for n in (s[1:3], s[3:5], s[5:7])])

class RGBA(traits.HasTraits):
    # r,g,b,a in the range 0-1 with default color 0,0,0,1 (black)
    r = traits.Range(0., 1., 0.)
    g = traits.Range(0., 1., 0.)
    b = traits.Range(0., 1., 0.)
    a = traits.Range(0., 1., 1.)

    def __init__(self, r=0., g=0., b=0., a=1.):
        self.r = r
        self.g = g
        self.b = b
        self.a = a

    def __repr__(self):
        return 'r,g,b,a = (%s, %s, %s, %s)\n\n(self.r, self.g, self.b, self.a)'

    def tuple_to_rgba(ob, name, val):
        tup = [float(x) for x in val]
        if len(tup)==3:
            r,g,b = tup
            return RGBA(r,g,b)
        elif len(tup)==4:
            r,g,b,a = tup
            return RGBA(r,g,b,a)
        else:
            raise ValueError

tuple_to_rgba.info = 'a RGB or RGBA tuple of floats'
```

def hex_to_rgba(ob, name, val):
    rgx = re.compile('^#[0-9A-Fa-f]{6}$')

    if not is_string_like(val):
        raise TypeError
    if rgx.match(val) is None:
        raise ValueError
    r, g, b = hex2color(val)
    return RGBA(r, g, b, 1.0)

hex_to_rgba.info = 'a hex color string'

def colorname_to_rgba(ob, name, val):
    hex = colors[val.lower()]
    r, g, b = hex2color(hex)
    return RGBA(r, g, b, 1.0)

colorname_to_rgba.info = 'a named color'

def float_to_rgba(ob, name, val):
    val = float(val)
    return RGBA(val, val, val, 1.)

float_to_rgba.info = 'a grayscale intensity'

Color = traits.Trait(RGBA(), float_to_rgba, colorname_to_rgba, RGBA,
                     hex_to_rgba, tuple_to_rgba)

def file_exists(ob, name, val):
    fh = file(val, 'r')
    return val

def path_exists(ob, name, val):
    os.path.exists(val)

TICKLEFT, TICKRIGHT, TICKUP, TICKDOWN = range(4)
linestyles = ('-', '--', '-.', ':', 'steps', 'None')

class LineRC(traits.HasTraits):
    linewidth = traits.Float(0.5)
    linestyle = traits.Trait(*linestyles)
    color = Color
    marker = traits.Trait(*linemarkers)
    markerfacecolor = Color
    markeredgecolor = Color
    markeredgewidth = traits.Float(0.5)
markersize = traits.Float(6)
anantialiased = flexible_true_trait
data_clipping = flexible_false_trait

```python
class PatchRC(traits.HasTraits):
    linewidth = traits.Float(1.0)
    facecolor = Color
    edgecolor = Color
    antialiased = flexible_true_trait

timezones = ["UTC", "US/Central", "ES/Eastern"] # fixme: and many more
backends = ('GTKAgg', 'Cairo', 'GDK', 'GTK', 'Agg',
            'GTKCairo', 'PS', 'SVG', 'Template', 'TkAgg',
            'WX')

class RC(traits.HasTraits):
    backend = traits.Trait(*backends)
    interactive = flexible_false_trait
    toolbar = traits.Trait('toolbar2', 'classic', None)
    timezone = traits.Trait(*timezones)
    lines = traits.Trait(LineRC())
    patch = traits.Trait(PatchRC())

rc = RC()
rc.lines.color = 'r'
if doprint:
    print('RC')
    rc.print_traits()
    print('RC lines')
    rc.lines.print_traits()
    print('RC patches')
    rc.patch.print_traits()
```

class Patch(Artist, traits.HasTraits):
    linewidth = traits.Float(0.5)
    facecolor = Color
    fc = facecolor
    edgcolor = Color
    fill = flexible_true_trait
    def __init__(self,
                 edgcolor=None,
                 facecolor=None,
                 linewdith=None,
                 antialiased = None,
                 fill=1,
                 **kwargs
                 ):
        Artist.__init__(self)
        if edgecolor is None: edgecolor = rc.patch.edgecolor
        if facecolor is None: facecolor = rc.patch.facecolor
        if linewidth is None: linewidth = rc.patch.linewidth
```python
if antialiased is None: antialiased = rc.patch.antialiased

self.edgecolor = edgecolor
self.facecolor = facecolor
self.linewidth = linewidth
self.antialiased = antialiased
self.fill = fill

p = Patch()
p.facecolor = '#bfbf00'
p.edgecolor = 'gold'
p.facecolor = (1,.5,.5,.25)
p.facecolor = 0.25
p.fill = 'f'
print(p.facecolor, type(p.facecolor), p.facecolor)
print(p.fill, type(p.fill), p.fill)
if p.fill_: print('fill')
else: print('no fill')
if doprint:
    print()
    print('Patch')
    p.print_traits()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 81.9 misc example code: rec_groupby_demo.py

[source code]

```python
from __future__ import print_function
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading', datafile)
r = mlab.csv2rec(datafile)
r.sort()

def daily_return(prices):
    'an array of daily returns from price array'
    g = np.zeros_like(prices)
    g[1:] = (prices[1:]-prices[:-1])/prices[:-1]
    return g

def volume_code(volume):
    'code the continuous volume data categorically'
    ind = np.searchsorted([1e5,1e6, 5e6,10e6, 1e7], volume)
    return ind
```

81.9. misc example code: rec_groupby_demo.py 1861
# a list of (dtype_name, summary_function, output_dtype_name).
# rec_summarize will call on each function on the indicated recarray
# attribute, and the result assigned to output name in the return
# record array.
summaries = [
    ('date', lambda x: [thisdate.year for thisdate in x], 'years'),
    ('date', lambda x: [thisdate.month for thisdate in x], 'months'),
    ('date', lambda x: [thisdate.weekday() for thisdate in x], 'weekday'),
    ('adj_close', daily_return, 'dreturn'),
    ('volume', volume_code, 'volcode'),
]

rsum = mlab.rec_summarize(r, summaries)

# stats is a list of (dtype_name, function, output_dtype_name).
# rec_groupby will summarize the attribute identified by the
# dtype_name over the groups in the groupby list, and assign the
# result to the output_dtype_name
stats = [
    ('dreturn', len, 'rcnt'),
    ('dreturn', np.mean, 'rmean'),
    ('dreturn', np.median, 'rmedian'),
    ('dreturn', np.std, 'rsigma'),
]

# you can summarize over a single variable, like years or months
print('summary by years')
ry = mlab.rec_groupby(rsum, ('years'), stats)
print(mlab.rec2txt(ry))

print('summary by months')
rm = mlab.rec_groupby(rsum, ('months'), stats)
print(mlab.rec2txt(rm))

# or over multiple variables like years and months
print('summary by year and month')
rym = mlab.rec_groupby(rsum, ('years', 'months'), stats)
print(mlab.rec2txt(rym))

print('summary by volume')
rv = mlab.rec_groupby(rsum, ('volcode'), stats)
print(mlab.rec2txt(rv))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.10 misc example code: rec_join_demo.py

[source code]

from __future__ import print_function
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading', datafile)
r = mlab.csv2rec(datafile)

r.sort()
r1 = r[-10:]

# Create a new array
r2 = np.empty(12, dtype=[('date', '|O4'), ('high', np.float),
                         ('marker', np.float)])
r2 = r2.view(np.recarray)
r2.date = r.date[-17:-5]
r2.high = r.high[-17:-5]
r2.marker = np.arange(12)

print("r1:"
print(mlab.rec2txt(r1))
print("r2:"
print(mlab.rec2txt(r2))

defaults = {'marker':-1, 'close':np.NaN, 'low':-4444.}

for s in ('inner', 'outer', 'leftouter'):
    rec = mlab.rec_join([['date', 'high']], r1, r2,
                         jointype=s, defaults=defaults)
    print("\n%sjoin :
%s" % (s, mlab.rec2txt(rec)))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

81.11 misc example code: sample_data_demo.py

[source code]

"""
Grab mpl data from the ~/.matplotlib/sample_data cache if it exists, else
fetch it from github and cache it
"""

from __future__ import print_function
import matplotlib.cbook as cbook
import matplotlib.pyplot as plt
fname = cbook.get_sample_data('ada.png', asfileobj=False)

print('fname', fname)
im = plt.imread(fname)
plt.imshow(im)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
81.12 misc example code: svg_filter_line.py

```python
from __future__ import print_function
import matplotlib
matplotlib.use("Svg")
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

fig1 = plt.figure()
ax = fig1.add_axes([0.1, 0.1, 0.8, 0.8])

# draw lines
l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-", mec="b", lw=5, ms=10, label="Line 1")
l2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "rs-", mec="r", lw=5, ms=10, color="r", label="Line 2")

for l in [l1, l2]:
    # draw shadows with same lines with slight offset and gray colors.
    xx = l.get_xdata()
    yy = l.get_ydata()
    shadow, = ax.plot(xx, yy)
    shadow.update_from(l)
    # adjust color
    shadow.set_color("0.2")
    # adjust zorder of the shadow lines so that it is drawn below the
    # original lines
    shadow.set_zorder(l.get_zorder()-0.5)
    # offset transform
    ot = mtransforms.offset_copy(l.get_transform(), fig1, x=4.0, y=-6.0, units='points')
    shadow.set_transform(ot)
    # set the id for a later use
    shadow.set_gid(l.get_label()+"_shadow")
```

Note that the filtering effects are only effective if your svg renderer support it.
ax.set_xlim(0., 1.)
ax.set_ylim(0., 1.)

# save the figure as a string in the svg format.
from StringIO import StringIO
f = StringIO()
plt.savefig(f, format="svg")

import xml.etree.cElementTree as ET

# filter definition for a gaussian blur
filter_def = ""
<defs xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/xlink'>
  <filter id='dropshadow' height='1.2' width='1.2'>
    <feGaussianBlur result='blur' stdDeviation='3'/>
  </filter>
</defs>

# read in the saved svg
tree, xmlid = ET.XMLID(f.getvalue())

# insert the filter definition in the svg dom tree.
tree.insert(0, ET.XML(filter_def))

for l in [l1, l2]:
    # pick up the svg element with given id
    shadow = xmlid[l.get_label()+"_shadow"]
    # apply shdow filter
    shadow.set("filter", 'url(#dropshadow)')

fn = "svg_filter_line.svg"
print("Saving '%s'" % fn)
ET.ElementTree(tree).write(fn)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib
matplotlib.use("Svg")

import matplotlib.pyplot as plt
from matplotlib.patches import Shadow

# make a square figure and axes
fig1 = plt.figure(1, figsize=(6,6))
ax = fig1.add_axes([0.1, 0.1, 0.8, 0.8])

labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
fracs = [15,30,45, 10]
expode=(0, 0.05, 0, 0)

# We want to draw the shadow for each pie but we will not use "shadow"
# option as it doesn't save the references to the shadow patches.
pies = ax.pie(fracs, explode=explode, labels=labels, autopct='%1.1f%%')

for w in pies[0]:
    # set the id with the label.
    w.set_gid(w.get_label())

    # we don't want to draw the edge of the pie
    w.set_ec("none")

for w in pies[0]:
    # create shadow patch
    s = Shadow(w, -0.01, -0.01)
    s.set_gid(w.get_gid()+"_shadow")
    s.set_zorder(w.get_zorder() - 0.1)
    ax.add_patch(s)

# save
from StringIO import StringIO
f = StringIO()
plt.savefig(f, format="svg")

import xml.etree.cElementTree as ET

# filter definition for shadow using a gaussian blur
# and lightening effect.
# The lightening filter is copied from http://www.w3.org/TR/SVG/filters.html

# I tested it with Inkscape and Firefox3. "Gaussian blur" is supported
# in both, but the lightening effect only in the inkscape. Also note
# that, inkscape's exporting also may not support it.

filter_def = ""
    <defs xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/xlink'>
<filter id='dropshadow' height='1.2' width='1.2'>
  <feGaussianBlur result='blur' stdDeviation='2'/>
</filter>

<filter id='MyFilter' filterUnits='objectBoundingBox' x='0' y='0' width='1' height='1'>
  <feGaussianBlur in='SourceAlpha' stdDeviation='4%' result='blur'/>
  <feOffset in='blur' dx='4%' dy='4%' result='offsetBlur'/>
  <feSpecularLighting in='blur' surfaceScale='5' specularConstant='.75'
    specularExponent='20' lighting-color="#bbbbbb" result='specOut'>
    <fePointLight x='-5000%' y='-10000%' z='20000%' />
  </feSpecularLighting>
  <feComposite in='specOut' in2='SourceAlpha' operator='in' result='specOut'/>
  <feComposite in='SourceGraphic' in2='specOut' operator='arithmetic'
    k1='0' k2='1' k3='1' k4='0'/>
</filter>
</defs>

```
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes([0.1, 0.3, 0.5, 0.5])
ax.pcolormesh(np.array([[1,2],[3,4]]))
plt.yticks([0.5, 1.5], ["long long tick label", "tick label"])
plt.ylabel("My y-label")
```
plt.title("Check saved figures for their bboxes")
for ext in ["png", "pdf", "svg", "svgz", "eps"]:
    print("saving tight_bbox_test.%s" % (ext,))
    plt.savefig("tight_bbox_test.%s" % (ext,), bbox_inches="tight")
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
82.1 mplot3d example code: 2dcollections3d_demo.py

```python
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca(projection='3d')

x = np.linspace(0, 1, 100)
```

---

**CHAPTER EIGHTY-TWO**

**MPLTO3D EXAMPLES**

82.1 mplot3d example code: 2dcollections3d_demo.py

```python
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca(projection='3d')

x = np.linspace(0, 1, 100)
```
```
y = np.sin(x * 2 * np.pi) / 2 + 0.5
ax.plot(x, y, zs=0, zdir='z', label='zs=0, zdir=z')

colors = ('r', 'g', 'b', 'k')
for c in colors:
    x = np.random.sample(20)
    y = np.random.sample(20)
    ax.scatter(x, y, 0, zdir='y', c=c)

ax.legend()
ax.set_xlim3d(0, 1)
ax.set_ylim3d(0, 1)
ax.set_zlim3d(0, 1)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```

82.2 mplot3d example code: bars3d_demo.py
```python
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
    xs = np.arange(20)
    ys = np.random.rand(20)
    # You can provide either a single color or an array. To demonstrate this,
    # the first bar of each set will be colored cyan.
    cs = [c] * len(xs)
    cs[0] = 'c'
    ax.bar(xs, ys, zs=z, zdir='y', color=cs, alpha=0.8)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```

82.3 mplot3d example code: contour3d_demo.py
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.4 mplot3d example code: contour3d_demo2.py
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, extend3d=True, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.5 mplot3d example code: contour3d_demo3.py

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contour(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)
```
```python
ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.6 mplot3d example code: contourf3d_demo.py

```
.. versionadded:: 1.1.0

This demo depends on new features added to contourf3d.

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x, y = np.random.rand(2, 100) * 4
hist, xedges, yedges = np.histogram2d(x, y, bins=4)

hist, xedges, yedges = np.histogram2d(x, y, bins=4)
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

82.9 mplot3d example code: lines3d_demo.py
mpl.rcParams['legend.fontsize'] = 10

fig = plt.figure()
ax = fig.gca(projection='3d')
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z, label='parametric curve')
ax.legend()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.10 mplot3d example code: lorenz_attractor.py

# Plot of the Lorenz Attractor based on Edward Lorenz's 1963 "Deterministic
# Nonperiodic Flow" publication.
# http://journals.ametsoc.org/doi/abs/10.1175/1520-0469%281963%29020%3C0130%3ADNF%3E2.0.CO%3B2
#
# Note: Because this is a simple non-linear ODE, it would be more easily
# done using SciPy's ode solver, but this approach depends only
# upon NumPy.

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def lorenz(x, y, z, s=10, r=28, b=2.667):
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return x_dot, y_dot, z_dot

dt = 0.01
stepCnt = 10000

# Need one more for the initial values
xs = np.empty((stepCnt + 1,))
ys = np.empty((stepCnt + 1,))
zs = np.empty((stepCnt + 1,))

# Setting initial values
xs[0], ys[0], zs[0] = (0., 1., 1.05)

# Stepping through "time".
for i in range(stepCnt):
    # Derivatives of the X, Y, Z state
    x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
    xs[i + 1] = xs[i] + (x_dot * dt)
    ys[i + 1] = ys[i] + (y_dot * dt)
    zs[i + 1] = zs[i] + (z_dot * dt)

fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot(xs, ys, zs)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
A tale of 2 subplots

Damped oscillation
Demonstrate the mixing of 2d and 3d subplots

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    s1 = np.cos(2*np.pi*t)
    e1 = np.exp(-t)
    return np.multiply(s1,e1)

#################
# First subplot
#################

# Twice as tall as it is wide.
fig = plt.figure(figsize=plt.figaspect(2.))
fig.suptitle('A tale of 2 subplots')
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

l = ax.plot(t1, f(t1), 'bo', t2, f(t2), 'k--', markerfacecolor='green')
ax.grid(True)
as.set_ylabel('Damped oscillation')

# Second subplot
ax = fig.add_subplot(2, 1, 2, projection='3d')
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                        linewidth=0, antialiased=False)

ax.set_zlim3d(-1, 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

# This example demonstrates mplot3d's offset text display.
# As one rotates the 3D figure, the offsets should remain oriented
# same way as the axis label, and should also be located "away"
# from the center of the plot.

# This demo triggers the display of the offset text for the x and
# y axis by adding 1e5 to X and Y. Anything less would not
# automatically trigger it.

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y = np.mgrid[0:6*np.pi:0.25, 0:4*np.pi:0.25]
Z = np.sqrt(np.abs(np.cos(X) + np.cos(Y)))
surf = ax.plot_surface(X + 1e5, Y + 1e5, Z, cmap='autumn', cstride=2, rstride=2)
ax.set_xlabel("X-Label")
ax.set_ylabel("Y-Label")
ax.set_zlabel("Z-Label")
ax.set_zlim(0, 2)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.13 mplot3d example code: pathpatch3d_demo.py

```python
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, PathPatch
# register Axes3D class with matplotlib by importing Axes3D
from mpl_toolkits.mplot3d import Axes3D
import mpl_toolkits.mplot3d.art3d as art3d
from matplotlib.text import TextPath
from matplotlib.transforms import Affine2D

def text3d(ax, xyz, s, zdir="z", size=None, angle=0, usetex=False, **kwargs):
    x, y, z = xyz
    if zdir == "y":
        xy1, z1 = (x, z), y
```

82.13. mplot3d example code: pathpatch3d_demo.py 1883
**elif** zdir == "y":
    xy1, z1 = (y, z), x
**else**:
    xy1, z1 = (x, y), z

text_path = TextPath((0, 0), s, size=size, usetex=usetex)
trans = Affine2D().rotate(angle).translate(xy1[0], xy1[1])

pl = PathPatch(trans.transform_path(text_path), **kwargs)
ap.add_patch(pl)
art3d.pathpatch_2d_to_3d(pl, z=z1, zdir=zdir)

fig = plt.figure()
ap = fig.add_subplot(111, projection='3d')
p = Circle((5, 5), 3)
ap.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z=0, zdir="x")

text3d(ax, (4, -2, 0), "X-axis", zdir="z", size=.5, usetex=False,
        ec="none", fc="k")
text3d(ax, (12, 4, 0), "Y-axis", zdir="z", size=.5, usetex=False,
        angle=.5*3.14159, ec="none", fc="k")
text3d(ax, (12, 10, 4), "Z-axis", zdir="y", size=.5, usetex=False,
        angle=.5*3.14159, ec="none", fc="k")

text3d(ax, (1, 5, 0),
        r"\displaystyle G_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu} $",
        zdir="z", size=1, usetex=True,
        ec="none", fc="k")

ax.set_xlim3d(0, 10)
ax.set_ylim3d(0, 10)
ax.set_zlim3d(0, 10)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
from matplotlib.colors import colorConverter
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

cc = lambda arg: colorConverter.to_rgba(arg, alpha=0.6)
xs = np.arange(0, 10, 0.4)
verts = []
zs = [0.0, 1.0, 2.0, 3.0]
for z in zs:
    ys = np.random.rand(len(xs))
    ys[0], ys[-1] = 0, 0
    verts.append(list(zip(xs, ys)))

poly = PolyCollection(verts, facecolors = [cc('r'), cc('g'), cc('b'), cc('y')])
poly.set_alpha(0.7)
```python
ax.add_collection3d(poly, zs=zs, zdir='y')

ax.set_xlabel('X')
ax.set_xlim3d(0, 10)
ax.set_ylabel('Y')
ax.set_ylim3d(-1, 4)
ax.set_zlabel('Z')
ax.set_zlim3d(0, 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.15 mplot3d example code: quiver3d_demo.py

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

plt.show()
```

1886 Chapter 82. mplot3d Examples
x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
    np.arange(-0.8, 1, 0.2),
    np.arange(-0.8, 1, 0.8))

u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0) * np.cos(np.pi * x) * np.cos(np.pi * y) *
    np.sin(np.pi * z))

ax.quiver(x, y, z, u, v, w, length=0.1)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.16  mplot3d example code: rotate_axes3d_demo.py
```python
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.1)
ax.plot_wireframe(X, Y, Z, rstride=5, cstride=5)

for angle in range(0, 360):
    ax.view_init(30, angle)
    plt.draw()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.17 mplot3d example code: scatter3d_demo.py

```
n = 100
for c, m, zl, zh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zl, zh)
    ax.scatter(xs, ys, zs, c=c, marker=m)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.18 mplot3d example code: subplot3d_demo.py

from mpl_toolkits.mplot3d.axes3d import Axes3D
import matplotlib.pyplot as plt

# imports specific to the plots in this example
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import get_test_data

# Twice as wide as it is tall.
fig = plt.figure(figsize=plt.figaspect(0.5))

#---- First subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
linewidth=0, antialiased=False)
ax.set_zlim3d(-1.01, 1.01)
fig.colorbar(surf, shrink=0.5, aspect=10)

#---- Second subplot
ax = fig.add_subplot(1, 2, 2, projection='3d')
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.19 mplot3d example code: surface3d_demo.py
```python
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
al = fig.add_subplot(111, projection='3d')

u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)

x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))

al.plot_surface(x, y, z, rstride=4, cstride=4, color='b')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

colortuple = ('y', 'b')
colors = np.empty(X.shape, dtype=str)
for y in range(ylen):
    for x in range(xlen):
        colors[x, y] = colortuple[(x + y) % len(colortuple)]
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, facecolors=colors,
linewidth=0, antialiased=False)

ax.set_zlim3d(-1, 1)
ax.w_zaxis.set_major_locator(LinearLocator(6))

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.22 mplot3d example code: surface3d_radial_demo.py

# By Armin Moser

from mpl_toolkits.mplot3d import Axes3D
import matplotlib
import numpy as np
from matplotlib import cm
from matplotlib import pyplot as plt
step = 0.04
maxval = 1.0
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# create supporting points in polar coordinates
r = np.linspace(0,1.25,50)
p = np.linspace(0,2*np.pi,50)
R, P = np.meshgrid(r, p)

# transform them to cartesian system
X, Y = R*np.cos(P), R*np.sin(P)
Z = ((R**2 - 1)**2)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.YlGnBu_r)
ax.set_zlim3d(0, 1)
ax.set_xlabel(r'$\phi_{\mathrm{real}}$')
ax.set_ylabel(r'$\phi_{\mathrm{im}}$')
ax.set_zlabel(r'$V(\phi)$')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.23 mplot3d example code: text3d_demo.py
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca(projection='3d')

zdirs = (None, 'x', 'y', 'z', (1, 1, 0), (1, 1, 1))
xs = (1, 4, 4, 9, 4, 1)
ys = (2, 5, 8, 10, 1, 2)
zs = (10, 3, 8, 9, 1, 8)

for zdir, x, y, z in zip(zdirs, xs, ys, zs):
    label = '(%d, %d, %d), dir=%s' % (x, y, z, zdir)
    ax.text(x, y, z, label, zdir)

ax.text(9, 0, 0, "red", color='red')
ax.text2D(0.05, 0.95, "2D Text", transform=ax.transAxes)

ax.set_xlim3d(0, 10)
ax.set_ylim3d(0, 10)
ax.set_zlim3d(0, 10)

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as tri
import numpy as np
import math

# First create the x and y coordinates of the points.
angles = 48
radius = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[...,np.newaxis], n_radii, axis=1)
angles[1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()
# Create a custom triangulation
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

plt.figure()
plt.gca(projection='3d')
plt.tricontour(triang, z)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.25 mplot3d example code: tricontourf3d_demo.py

Contour plots of unstructured triangular grids.

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as tri
import numpy as np
import math

# First create the x and y coordinates of the points.
n_angles = 48
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[...,np.newaxis], n_radii, axis=1)
angles[:,1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create a custom triangulation
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

plt.figure()
plt.gca(projection='3d')
plt.tricontourf(triang, z)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

n_angles = 36
n_radii = 8

# An array of radii
# Does not include radius r=0, this is to eliminate duplicate points
radii = np.linspace(0.125, 1.0, n_radii)

# An array of angles
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)

# Repeat all angles for each radius
angles = np.repeat(angles[...,np.newaxis], n_radii, axis=1)

# Convert polar (radii, angles) coords to cartesian (x, y) coords
# (0, 0) is added here. There are no duplicate points in the (x, y) plane
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())
# PrINGLE sURFACE

```python
z = np.sin(-x*y)

fig = plt.figure()
ax = fig.gca(projection='3d')

tax.plot_trisurf(x, y, z, cmap=cm.jet, linewidth=0.2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.27 mplot3d example code: trisurf3d_demo2.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as mtri

# u, v are parameterisation variables
u = (np.linspace(0, 2.0 * np.pi, endpoint=True, num=50) * np.ones((10, 1))).flatten()
v = np.repeat(np.linspace(-0.5, 0.5, endpoint=True, num=10), repeats=50).flatten()

# This is the Mobius mapping, taking a u, v pair and returning an x, y, z
# triple
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space to determine the triangles
tri = mtri.Triangulation(u, v)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')

# The triangles in parameter space determine which x, y, z points are
# connected by an edge
ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap=plt.cm.Spectral)
ax.set_zlim(-1, 1)

# First create the x and y coordinates of the points.

n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[... ,np.newaxis], n_radii, axis=1)
angles[::1,::2] += np.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = mtri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# triplot plot.
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.plot_trisurf(triang, z, cmap=plt.cm.CMRmap)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from __future__ import print_function

"""
A very simple 'animation' of a 3D plot
"""

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
import time

def generate(X, Y, phi):
    R = 1 - np.sqrt(X**2 + Y**2)
    return np.cos(2 * np.pi * X + phi) * R

plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

xs = np.linspace(-1, 1, 50)
ys = np.linspace(-1, 1, 50)
X, Y = np.meshgrid(xs, ys)
Z = generate(X, Y, 0.0)
wframe = None
start = time.time()
for phi in np.linspace(0, 360 / 2 / np.pi, 100):
    oldcol = wframe
    Z = generate(X, Y, phi)
    wframe = ax.plot_wireframe(X, Y, Z, rstride=2, cstride=2)

    # Remove old line collection before drawing
    if oldcol is not None:
        ax.collections.remove(oldcol)

    plt.pause(.001)

print('FPS: %f' % (100 / (time.time() - start)))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

82.29 mplot3d example code: wire3d_demo.py
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
83.1 pie_and_polar_charts example code: pie_demo_features.py

Demo of a basic pie chart plus a few additional features.

In addition to the basic pie chart, this demo shows a few optional features:

* slice labels
* auto-labeling the percentage
* offsetting a slice with "explode"
Note about the custom start angle:

The default `"startangle"` is 0, which would start the "Frogs" slice on the positive x-axis. This example sets `"startangle = 90"` such that everything is rotated counter-clockwise by 90 degrees, and the frog slice starts on the positive y-axis.

```python
import matplotlib.pyplot as plt

# The slices will be ordered and plotted counter-clockwise.
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
explode = (0, 0.1, 0, 0) # only "explode" the 2nd slice (i.e. 'Hogs')

plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%.1f%%', shadow=True, startangle=90)

plt.axis('equal')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
83.2 pie_and_polarCharts example code: polar_bar_demo.py

```
import numpy as np
import matplotlib.pyplot as plt

N = 20
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)

ax = plt.subplot(111, polar=True)
bars = ax.bar(theta, radii, width=width, bottom=0.0)

# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)

plt.show()
```

---

83.2. pie_and_polarCharts example code: polar_bar_demo.py 1909
Keywords: python, matplotlib, pylab, example, codex (see Search examples)

83.3 pie_and_polar_charts example code: polar_scatter_demo.py

Demo of scatter plot on a polar axis.

Size increases radially in this example and color increases with angle (just to verify the symbols are being scattered correctly).

```python
import numpy as np
import matplotlib.pyplot as plt

N = 150
r = 2 * np.random.rand(N)
theta = 2 * np.pi * np.random.rand(N)
area = 200 * r**2 * np.random.rand(N)
colors = theta

ax = plt.subplot(111, polar=True)
c = plt.scatter(theta, r, c=colors, s=area, cmap=plt.cm.hsv)
c.set_alpha(0.75)
```
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.1 pylab_examples example code: accented_text.py

```python
#!/usr/bin/env python

"""
matplotlib supports accented characters via TeX mathtext

The following accents are provided: \hat, \breve, \grave, \bar, \acute, \tilde, \vec, \dot, \ddot. All of them have the same syntax, e.g., to make an overbar you do \bar{o} or to make an o umlaut you do \ddot{o}. The shortcuts are also provided, e.g.,: \"o \text{ e} \text{ e} \text{ ~ n} \text{ x}
"""
```
```python
from pylab import *

axes([0.1, 0.15, 0.8, 0.75])
plot(range(10))

title(r'$\ddot{o}\acute{e}\grave{e}\hat{O}\breve{i}\bar{A}\tilde{n}\vec{q}$', fontsize=20)
# shorthand is also supported and curly's are optional
xlabel(r'''$"o\ddot o \ e\`e\~n\e\^y$''', fontsize=20)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 84.2 `pylab_examples` example code: `agg_buffer_to_array.py`
import matplotlib.pyplot as plt
import numpy as np

# make an agg figure
fig, ax = plt.subplots()
ax.plot([1,2,3])
ax.set_title('a simple figure')
fig.canvas.draw()

# grab the pixel buffer and dump it into a numpy array
buf = fig.canvas.buffer_rgba()
l, b, w, h = fig.bbox.bounds
# The array needs to be copied, because the underlying buffer
# may be reallocated when the window is resized.
X = np.frombuffer(buf, np.uint8).copy()
X.shape = h,w,4

# now display the array X as an Axes in a new figure
fig2 = plt.figure()
ax2 = fig2.add_subplot(111, frameon=False)
ax2.imshow(X)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
You can precisely layout text in data or axes (0,1) coordinates. This example shows you some of the alignment and rotation specifications to layout text.

```python
# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height
ax = gca()
p = Rectangle((left, bottom), width, height,
              fill=False,
             )
p.set_transform(ax.transAxes)
p.set_clip_on(False)
```
ax.add_patch(p)

ax.text(left, bottom, 'left top',
       horizontalalignment='left',
       verticalalignment='top',
       transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
       horizontalalignment='left',
       verticalalignment='bottom',
       transform=ax.transAxes)

ax.text(right, top, 'right bottom',
       horizontalalignment='right',
       verticalalignment='bottom',
       transform=ax.transAxes)

ax.text(right, top, 'right top',
       horizontalalignment='right',
       verticalalignment='top',
       transform=ax.transAxes)

ax.text(right, bottom, 'center top',
       horizontalalignment='center',
       verticalalignment='top',
       transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
       horizontalalignment='right',
       verticalalignment='center',
       rotation='vertical',
       transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
       horizontalalignment='left',
       verticalalignment='center',
       rotation='vertical',
       transform=ax.transAxes)

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
       horizontalalignment='center',
       verticalalignment='center',
       transform=ax.transAxes)

ax.text(right, 0.5*(bottom+top), 'centered',
       horizontalalignment='center',
       verticalalignment='center',
       rotation='vertical',
       transform=ax.transAxes)

ax.text(left, top, 'rotated
with newlines',
       horizontalalignment='center',
       verticalalignment='center',
       rotation='vertical',
       transform=ax.transAxes)
verticalalignment='center', rotation=45, transform=ax.transAxes)

axis('off')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.4 pylab_examples example code: anchored_artists.py

from matplotlib.patches import Rectangle, Ellipse

from matplotlib.offsetbox import AnchoredOffsetbox, AuxTransformBox, VPacker, \n    TextArea, DrawingArea

class AnchoredText(AnchoredOffsetbox):
    def __init__(self, s, loc, pad=0.4, borderpad=0.5, prop=None, frameon=True):

        self.txt = TextArea(s,
minimumdescent=False)

super(AnchoredText, self).__init__(loc, pad=pad, borderpad=borderpad,
               child=self.txt,
               prop=prop,
               frameon=frameon)

class AnchoredSizeBar(AnchoredOffsetbox):
    def __init__(self, transform, size, label, loc,
               pad=0.1, borderpad=0.1, sep=2, prop=None, frameon=True):
        ""
        Draw a horizontal bar with the size in data coordinate of the give axes.
        A label will be drawn underneath (center-aligned).

        pad, borderpad in fraction of the legend font size (or prop)
        sep in points.
        ""
        self.size_bar = AuxTransformBox(transform)
        self.size_bar.add_artist(Rectangle((0,0), size, 0, fc="none"))

        self.txt_label = TextArea(label, minimumdescent=False)

        self._box = VPacker(children=[self.size_bar, self.txt_label],
                  align="center",
                  pad=0, sep=sep)

        AnchoredOffsetbox.__init__(self, loc, pad=pad, borderpad=borderpad,
                         child=self._box,
                         prop=prop,
                         frameon=frameon)

class AnchoredEllipse(AnchoredOffsetbox):
    def __init__(self, transform, width, height, angle, loc,
               pad=0.1, borderpad=0.1, prop=None, frameon=True):
        ""
        Draw an ellipse the size in data coordinate of the give axes.

        pad, borderpad in fraction of the legend font size (or prop)
        ""
        self._box = AuxTransformBox(transform)
        self.ellipse = Ellipse((0,0), width, height, angle)
        self._box.add_artist(self.ellipse)

        AnchoredOffsetbox.__init__(self, loc, pad=pad, borderpad=borderpad,
                         child=self._box,
                         prop=prop,
                         frameon=frameon)
class AnchoredDrawingArea(AnchoredOffsetbox):
    def __init__(self, width, height, xdescent, ydescent, loc, pad=0.4, borderpad=0.5, prop=None, frameon=True):
        self.da = DrawingArea(width, height, xdescent, ydescent, clip=True)
        super(AnchoredDrawingArea, self).__init__(loc, pad=pad, borderpad=borderpad, child=self.da, prop=prop, frameon=frameon)

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    ax = plt.gca()
    ax.set_aspect(1.)
    at = AnchoredText("Figure 1a", loc=2, frameon=True)
    at.patch.set_boxstyle("round", pad=0., rounding_size=0.2)
    ax.add_artist(at)
    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0, loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0., loc=3, pad=0.5, borderpad=0.4, frameon=True)
    ax.add_artist(ae)
    # draw a horizontal bar with length of 0.1 in Data coordinate # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData, 0.1, r"1$^{\prime}$", loc=8, pad=0.1, borderpad=0.5, sep=5, frameon=False)
    ax.add_artist(asb)
    plt.draw()
    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Pyplot animation example.

The method shown here is only for very simple, low-performance use. For more demanding applications, look at the animation module and the examples that use it.

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(6)
y = np.arange(5)
z = x * y[:, np.newaxis]

for i in xrange(5):
    if i == 0:
        p = plt.imshow(z)
        fig = plt.gcf()
        plt.clim()  # clamp the color limits
        plt.title("Boring slide show")
    else:
```

Boring slide show

0 1 2 3 4 5
0
1
2
3
4
Boring slide show
```
z = z + 2
p.set_data(z)

print("step", i)
plt.pause(0.5)
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.6 pylab_examples example code: annotation_demo.py
a polar annotation
Some examples of how to annotate points in figures. You specify an
annotation point xy=(x,y) and a text point xytext=(x,y) for the
annotated points and text location, respectively. Optionally, you can
specify the coordinate system of xy and xytext with one of the
following strings for xycoords and textcoords (default is 'data')

'figure points'  : points from the lower left corner of the figure
'figure pixels'  : pixels from the lower left corner of the figure
'figure fraction': 0,0 is lower left of figure and 1,1 is upper, right
'axes points'    : points from lower left corner of axes
'axes pixels'    : pixels from lower left corner of axes
'axes fraction'  : 0,0 is lower left of axes and 1,1 is upper right
'offset points'  : Specify an offset (in points) from the xy value
'data'           : use the axes data coordinate system

Optionally, you can specify arrow properties which draws and arrow
from the text to the annotated point by giving a dictionary of arrow
properties

Valid keys are

width : the width of the arrow in points
frac : the fraction of the arrow length occupied by the head
headwidth : the width of the base of the arrow head in points
shrink : move the tip and base some percent away from the
annotated point and text
any key for matplotlib.patches.polygon (e.g., facecolor)

For physical coordinate systems (points or pixels) the origin is the
(bottom, left) of the figure or axes. If the value is negative, however, the origin is from the (right, top) of the figure or axes,
analogous to negative indexing of sequences.

```python
from matplotlib.pyplot import figure, show
from matplotlib.patches import Ellipse
import numpy as np

if 1:
    # if only one location is given, the text and xypoint being
    # annotated are assumed to be the same
    fig = figure()
    ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1,5), ylim=(-3,5))

    t = np.arange(0.0, 5.0, 0.01)
    s = np.cos(2*np.pi*t)
    line, = ax.plot(t, s, lw=3, color='purple')

    ax.annotate('axes center', xy=(.5, .5), xycoords='axes fraction',
                horizontalalignment='center', verticalalignment='center')

    ax.annotate('pixels', xy=(20, 20), xycoords='figure pixels')

    ax.annotate('points', xy=(100, 300), xycoords='figure points')

    ax.annotate('offset', xy=(1, 1), xycoords='data',
                xytext=(-15, 10), textcoords='offset points',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='right', verticalalignment='bottom',)

    ax.annotate('local max', xy=(3, 1), xycoords='data',
                xytext=(0.8, 0.95), textcoords='axes fraction',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='right', verticalalignment='top',)

    ax.annotate('a fractional title', xy=(.025, .975),
                xycoords='figure fraction',
                horizontalalignment='left', verticalalignment='top',
                fontsize=20)

    # use negative points or pixels to specify from right, top -10, 10
    # is 10 points to the left of the right side of the axes and 10
```
Matplotlib, Release 1.4.3

# points above the bottom
ax.annotate('bottom right (points)', xy=(-10, 10),
            xycoords='axes points',
            horizontalalignment='right', verticalalignment='bottom',
            fontsize=20)

if 1:
    # you can specify the xypoint and the xytext in different
    # positions and coordinate systems, and optionally turn on a
    # connecting line and mark the point with a marker. Annotations
    # work on polar axes too. In the example below, the xy point is
    # in native coordinates (xycoords defaults to 'data'). For a
    # polar axes, this is in (theta, radius) space. The text in this
    # example is placed in the fractional figure coordinate system.
    # Text keyword args like horizontal and vertical alignment are
    # respected
    fig = figure()
    ax = fig.add_subplot(111, polar=True)
    r = np.arange(0, 1, 0.001)
    theta = 2*np.pi*r
    line, = ax.plot(theta, r, color='#ee8d18', lw=3)
    ind = 800
    thisr, thistheta = r[ind], theta[ind]
    ax.plot([thistheta], [thisr], 'o')
    ax.annotate('a polar annotation',
                xy=(thistheta, thisr),  # theta, radius
                xytext=(0.05, 0.05),   # fraction, fraction
                textcoords='figure fraction',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='left',
                verticalalignment='bottom',
                )

if 1:
    # You can also use polar notation on a cartesian axes. Here the
    # native coordinate system ('data') is cartesian, so you need to
    # specify the xysrcds and textcoords as 'polar' if you want to
    # use (theta, radius)
    el = Ellipse((0, 0), 10, 20, facecolor='r', alpha=0.5)
    fig = figure()
    ax = fig.add_subplot(111, aspect='equal')
    ax.add_artist(el)
    el.set_clip_box(ax.bbox)
    ax.annotate('the top',
                xy=(np.pi/2., 10.),          # theta, radius
                xytext=(np.pi/3, 20.),      # theta, radius
                xycoords='polar',
                textcoords='polar',
                )
arrowprops=dict(facecolor='black', shrink=0.05),
horizontalalignment='left',
verticalalignment='bottom',
clip_on=True, # clip to the axes bounding box
)
ax.set_xlim(-20, 20)
ax.set_ylim(-20, 20)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.7 pylab_examples example code: annotation_demo2.py
Matplotlib, Release 1.4.3

3
2
1

fancy

0

wedge
wedge

1
2
3

simple

4

wedge

5 −>
1

0

1

2

3

4

5

from matplotlib.pyplot import figure, show
from matplotlib.patches import Ellipse
import numpy as np
if 1:
fig = figure(1,figsize=(8,5))
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1,5), ylim=(-4,3))
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=3, color='purple')
ax.annotate('arrowstyle', xy=(0, 1), xycoords='data',
xytext=(-50, 30), textcoords='offset points',
arrowprops=dict(arrowstyle="->")
)
ax.annotate('arc3', xy=(0.5, -1), xycoords='data',
xytext=(-30, -30), textcoords='offset points',
arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=.2")
)
ax.annotate('arc', xy=(1., 1), xycoords='data',
xytext=(-40, 30), textcoords='offset points',

1928

Chapter 84. pylab_examples Examples


arrowprops=dict(arrowstyle="->",
              connectionstyle="arc,angleA=0,armA=30,rad=10"),
          )

ax.annotate('arc', xy=(1.5, -1), xycoords='data',
            xytext=(-40, -30), textcoords='offset points',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc,angleA=0,armA=20,angleB=-90,armB=15,rad=7"),
          )

ax.annotate('angle', xy=(2., 1), xycoords='data',
            xytext=(-50, 30), textcoords='offset points',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle,angleA=0,angleB=90,rad=10"),
          )

ax.annotate('angle3', xy=(2.5, -1), xycoords='data',
            xytext=(-50, -30), textcoords='offset points',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle3,angleA=0,angleB=-90"),
          )

ax.annotate('angle', xy=(3., 1), xycoords='data',
            xytext=(-50, 30), textcoords='offset points',
            bbox=dict(boxstyle="round", fc="0.8"),
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle,angleA=0,angleB=90,rad=10"),
          )

ax.annotate('angle', xy=(3.5, -1), xycoords='data',
            xytext=(-70, -60), textcoords='offset points',
            size=20,
            bbox=dict(boxstyle="round4,pad=.5", fc="0.8"),
            arrowprops=dict(arrowstyle="->", shrinkA=0, shrinkB=10,
                            connectionstyle="angle,angleA=0,angleB=-90,rad=10"),
          )

ann = ax.annotate('', xy=(4., 1.), xycoords='data',
            xytext=(4.5, -1), textcoords='data',
            arrowprops=dict(arrowstyle="<->", shrinkA=0, shrinkB=10,
                            connectionstyle="bar", ec="k"),
            )

84.7. pylab_examples example code: annotation_demo2.py 1929
Matplotlib, Release 1.4.3

)
)

if 1:
fig = figure(2)
fig.clf()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1,5), ylim=(-5,3))
el = Ellipse((2, -1), 0.5, 0.5)
ax.add_patch(el)
ax.annotate('$->$', xy=(2., -1), xycoords='data',
xytext=(-150, -140), textcoords='offset points',
bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="->",
patchB=el,
connectionstyle="angle,angleA=90,angleB=0,rad=10"),
)
ax.annotate('fancy', xy=(2., -1), xycoords='data',
xytext=(-100, 60), textcoords='offset points',
size=20,
#bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="fancy",
fc="0.6", ec="none",
patchB=el,
connectionstyle="angle3,angleA=0,angleB=-90"),
)
ax.annotate('simple', xy=(2., -1), xycoords='data',
xytext=(100, 60), textcoords='offset points',
size=20,
#bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="simple",
fc="0.6", ec="none",
patchB=el,
connectionstyle="arc3,rad=0.3"),
)
ax.annotate('wedge', xy=(2., -1), xycoords='data',
xytext=(-100, -100), textcoords='offset points',
size=20,
#bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="wedge,tail_width=0.7",
fc="0.6", ec="none",
patchB=el,
connectionstyle="arc3,rad=-0.3"),
)

ann = ax.annotate('wedge', xy=(2., -1), xycoords='data',
xytext=(0, -45), textcoords='offset points',

1930

Chapter 84. pylab_examples Examples


size=20,
bbox=dict(boxstyle="round", fc=(1.0, 0.7, 0.7), ec=(1., .5, .5)),
arrowprops=dict(arrowstyle="wedge,tail_width=1.",
    fc=(1.0, 0.7, 0.7), ec=(1., .5, .5),
    patchA=None,
    patchB=el,
    relpos=(0.2, 0.8),
    connectionstyle="arc3,rad=-0.1"),
)

ann = ax.annotate('wedge', xy=(2., -1), xycoords='data',
    xytext=(35, 0), textcoords='offset points',
    size=20, va="center",
    bbox=dict(boxstyle="round", fc=(1.0, 0.7, 0.7), ec="none"),
    arrowprops=dict(arrowstyle="wedge,tail_width=1.",
        fc=(1.0, 0.7, 0.7), ec="none",
        patchA=None,
        patchB=el,
        relpos=(0.2, 0.5),
    )
)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)

bbox_args = dict(boxstyle="round", fc="0.8")
arrow_args = dict(arrowstyle="->")

ax1.annotate('figure fraction : 0, 0', xy=(0, 0), xycoords='figure fraction',
              xytext=(20, 20), textcoords='offset points',
              ha="left", va="bottom",
              bbox=bbox_args,
              arrowprops=arrow_args)

ax1.annotate('figure fraction : 1, 1', xy=(1, 1), xycoords='figure fraction',
              xytext=(-20, -20), textcoords='offset points',
              ha="right", va="top",
              bbox=bbox_args,
              arrowprops=arrow_args)

ax1.annotate('axes fraction : 0, 0', xy=(0, 0), xycoords='axes fraction',
              xytext=(0, 0))
```python
from matplotlib.text import OffsetFrom

an1 = ax1.annotate('Drag me 1', xy=(.5, .7), xycoords='data',
                     #xytext=(.5, .7), textcoords='data',
                     ha="center", va="center",
                     bbox=bbox_args,
                     #arrowprops=arrow_args
                  )

an2 = ax1.annotate('Drag me 2', xy=(.5, .5), xycoords=an1,
                     xytext=(.5, .3), textcoords='axes fraction',
                     ha="center", va="center",
                     bbox=bbox_args,
                     arrowprops=dict(patchB=an1.get_bbox_patch(),
                                      connectionstyle="arc3,rad=0.2",
                                      **arrow_args)
                  )

an3 = ax1.annotate('', xy=(.5, .5), xycoords=an2,
                     xytext=(.5, .5), textcoords=an1,
                     ha="center", va="center",
                     bbox=bbox_args,
                     arrowprops=dict(patchA=an1.get_bbox_patch(),
                                      patchB=an2.get_bbox_patch(),
                                      connectionstyle="arc3,rad=0.2",
                                      **arrow_args)
                  )

t = ax2.annotate('xy=(0, 1)
xycoords=('"data", "axes fraction")',
                 xy=(0, 1), xycoords=('"data", "axes fraction")',
                 xytext=(0, -20), textcoords='offset points',
                 ha="center", va="top",
                 bbox=bbox_args,
                 arrowprops=arrow_args
                )

from matplotlib.text import OffsetFrom

ax2.annotate('xy=(0.5, 0)\nxcoords=artist',
             )
```

84.8. `pylab_examples` example code: `annotation_demo3.py`
xy=(0.5, 0.), xycoords=t,
xytext=(0, -20), textcoords='offset points',
ha="center", va="top",
bbox=bbox_args,
arrowprops=arrow_args)

ax2.annotate('xy=(0.8, 0.5)\nxycoords=ax1.transData',
xy=(0.8, 0.5), xycoords=ax1.transData,
xytext=(10, 10), textcoords=OffsetFrom(ax2.bbox, (0, 0), "points"),
ha="left", va="bottom",
bbox=bbox_args,
arrowprops=arrow_args)

ax2.set_xlim(-2, 2)
ax2.set_ylim(-2, 2)

an1.draggable()
an2.draggable()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Edward Tufte uses this example from Anscombe to show 4 datasets of \( x \) and \( y \) that have the same mean, standard deviation, and regression line, but which are qualitatively different.

**matplotlib fun for a rainy day**

```python
#!/usr/bin/env python

from __future__ import print_function

"""
Edward Tufte uses this example from Anscombe to show 4 datasets of \( x \) and \( y \) that have the same mean, standard deviation, and regression line, but which are qualitatively different.

matplotlib fun for a rainy day
"""

from pylab import *

x = array([10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5])
y1 = array([8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68])
y2 = array([9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74])
y3 = array([7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73])
x4 = array([8,8,8,8,8,8,8,19,8,8,8])
y4 = array([6.58,5.76,7.71,8.84,8.47,7.04,5.25,12.50,5.56,7.91,6.89])

def fit(x):
    return 3+0.5*x
```
xfit = array([amin(x), amax(x)])

subplot(221)
plot(x, y1, 'ks', xfit, fit(xfit), 'r-', lw=2)
axis([2, 20, 2, 14])
setp(gca(), xticklabels=[], yticks=(4, 8, 12), xticks=(0, 10, 20))
text(3, 12, 'I', fontsize=20)

subplot(222)
plot(x, y2, 'ks', xfit, fit(xfit), 'r-', lw=2)
axis([2, 20, 2, 14])
setp(gca(), xticklabels=[], yticks=(4, 8, 12), yticklabels=[], xticks=(0, 10, 20))
text(3, 12, 'II', fontsize=20)

subplot(223)
plot(x, y3, 'ks', xfit, fit(xfit), 'r-', lw=2)
axis([2, 20, 2, 14])
text(3, 12, 'III', fontsize=20)
setp(gca(), yticks=(4, 8, 12), xticks=(0, 10, 20))

subplot(224)
xfit = array([amin(x4), amax(x4)])
plot(x4, y4, 'ks', xfit, fit(xfit), 'r-', lw=2)
axis([2, 20, 2, 14])
setp(gca(), yticklabels=[], yticks=(4, 8, 12), xticks=(0, 10, 20))
text(3, 12, 'IV', fontsize=20)

#verify the stats
pairs = (x, y1), (x, y2), (x, y3), (x4, y4)
for x, y in pairs:
    print('mean=%1.2f, std=%1.2f, r=%1.2f' % (mean(y), std(y), corrcoef(x, y)[0][1]))

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
from pylab import *

def f(t):
    'a damped exponential'
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return multiply(s1,e1)

t1 = arange(0.0, 5.0, .2)

l = plot(t1, f(t1), 'ro')
setp(l, 'markersize', 30)
setp(l, 'markerfacecolor', 'b')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""Arrow drawing example for the new fancy_arrow facilities.

Code contributed by: Rob Knight <rob@spot.colorado.edu>

usage:

   python arrow_demo.py realistic|full|sample|extreme

"""

from pylab import *

rates_to_bases={
    'r1': 'AT',
    'r2': 'TA',
    'r3': 'GA',
    'r4': 'AG',
    'r5': 'CA',
    'r6': 'AC',
    'r7': 'GT',
    'r8': 'TG',
    'r9': 'CT',
    'r10': 'TC',
    'r11': 'GC',
    'r12': 'CG'
}

numbered_bases_to_rates = dict(((v, k) for k, v in rates_to_bases.items()))
lettered_bases_to_rates = dict(((v, 'r' + v) for k, v in rates_to_bases.items()))

def add_dicts(d1, d2):
    """Adds two dicts and returns the result."""
    result = d1.copy()
    result.update(d2)
    return result

def make_arrow_plot(data, size=4, display='length', shape='right',
    max_arrow_width=0.03, arrow_sep = 0.02, alpha=0.5, 

normalizer_data=False, ec=None, labelcolor=None, \
head_starts_at_zero=True, rate_labels=lettered_bases_to_rates,\n**kwargs):
    """Makes an arrow plot.

Parameters:

data: dict with probabilities for the bases and pair transitions.
size: size of the graph in inches.
display: 'length', 'width', or 'alpha' for arrow property to change.
shape: full, left, or right for full or half arrows.
max_arrow_width: maximum width of an arrow, data coordinates.
arrow_sep: separation between arrows in a pair, data coordinates.
alpha: maximum opacity of arrows, default 0.8.

**kwargs can be anything allowed by a Arrow object, e.g.
linewidth and edgecolor.
""

xlim(-0.5,1.5)
ylim(-0.5,1.5)
gcf().set_size_inches(size,size)
xticks([])
yticks([])
max_text_size = size*12
min_text_size = size
label_text_size = size*2.5
text_params={'ha':'center', 'va':'center', 'family':'sans-serif',\
                'fontweight':'bold'}
r2 = sqrt(2)
deltas = {'AT':(1,0),
         'TA':(-1,0),
         'GA':(0,1),
         'AG':(0,-1),
         'CA':(-1/r2, 1/r2),
         'AC':(1/r2, -1/r2),
         'GT':(1/r2, 1/r2),
         'TG':(-1/r2,-1/r2),
         'CT':(0,1),
         'TC':(0,-1),
         'GC':(1,0),
         'CG':(-1,0)
}
colors = {'AT':r',
         'TA':k',
         'GA':g',
         'AG':r',
         'CA':b',
         'AC':r',
         'CT':b'
'GT': 'g',
'TG': 'k',
'CT': 'b',
'TC': 'k',
'GC': 'g',
'CG': 'b'}
label_positions = {
'AT': 'center',
'TA': 'center',
'GA': 'center',
'AG': 'center',
'CA': 'left',
'AC': 'left',
'GT': 'left',
'TG': 'left',
'CT': 'center',
'TC': 'center',
'GC': 'center',
'CG': 'center'
}

def do_fontsize(k):
    return float(clip(max_text_size*sqrt(data[k]),
    min_text_size,max_text_size))

A = text(0,1,'$A_3$', color='r', size=do_fontsize('A'), **text_params)
T = text(1,1,'$T_3$', color='k', size=do_fontsize('T'), **text_params)
G = text(0,0,'$G_3$', color='g', size=do_fontsize('G'), **text_params)
C = text(1,0,'$C_3$', color='b', size=do_fontsize('C'), **text_params)

arrow_h_offset = 0.25  # data coordinates, empirically determined
max_arrow_length = 1 - 2*arrow_h_offset
max_arrow_width = max_arrow_width
max_head_width = 2*max_arrow_width
max_head_length = 2*max_head_width
arrow_params={
    'length_includes_head':True, 'shape':shape,
    'head_starts_at_zero':head_starts_at_zero}
ax = gca()
sf = 0.6  # max arrow size represents this in data coords

d = (r2/2 + arrow_h_offset - 0.5)/r2  # distance for diags
r2v = arrow_sep/r2  # offset for diags

# tuple of x, y for start position
positions = {
    'AT': (arrow_h_offset, 1+arrow_sep),
    'TA': (1-arrow_h_offset, 1-arrow_sep),
    'GA': (-arrow_sep, arrow_h_offset),
    'AG': (arrow_sep, 1-arrow_h_offset),
if normalize_data:
    # find maximum value for rates, i.e. where keys are 2 chars long
    max_val = 0
    for k, v in data.items():
        if len(k) == 2:
            max_val = max(max_val, v)
    # divide rates by max val, multiply by arrow scale factor
    for k, v in data.items():
        data[k] = v/max_val*sf

def draw_arrow(pair, alpha=alpha, ec=ec, labelcolor=labelcolor):
    # set the length of the arrow
    if display == 'length':
        length = max_head_length+(max_arrow_length-max_head_length)*data[pair]/sf
        data[pair] /= sf
    else:
        length = max_arrow_length
    # set the transparency of the arrow
    if display == 'alph':
        alpha = min(data[pair]/sf, alpha)
    else:
        alpha=alpha
    # set the width of the arrow
    if display == 'width':
        scale = data[pair]/sf
        width = max_arrow_width*scale
        head_width = max_head_width*scale
        head_length = max_head_length*scale
    else:
        width = max_arrow_width
        head_width = max_head_width
        head_length = max_head_length

    fc = colors[pair]
    ec = ec or fc

    x_scale, y_scale = deltas[pair]
    x_pos, y_pos = positions[pair]
    arrow(x_pos, y_pos, x_scale*length, y_scale*length, 
          fc=fc, ec=ec, alpha=alpha, width=width, head_width=head_width, 
          head_length=head_length, **arrow_params)

    # figure out coordinates for text
# if drawing relative to base: x and y are same as for arrow
# dx and dy are one arrow width left and up
# need to rotate based on direction of arrow, use x_scale and y_scale
# as sin x and cos x?
sx, cx = y_scale, x_scale

where = label_positions[pair]
if where == 'left':
    orig_position = 3*array([[max_arrow_width, max_arrow_width]])
elif where == 'absolute':
    orig_position = array([[max_arrow_length/2.0, 3*max_arrow_width]])
elif where == 'right':
    orig_position = array([[length-3*max_arrow_width, \
                           3*max_arrow_width]])
elif where == 'center':
    orig_position = array([[length/2.0, 3*max_arrow_width]])
else:
    raise ValueError("Got unknown position parameter %s" % where)

M = array([[cx, sx],[-sx,cx]])
coords = dot(orig_position, M) + [[x_pos, y_pos]]
x, y = ravel(coords)
orig_label = rate_labels[pair]
label = '$%s_{-%{\mathrm{\text{mathrm}}(%s)}}$' % (orig_label[0], orig_label[1:]),
        color=labelcolor or fc)

text(x, y, label, size=label_text_size, ha='center', va='center', \
       color=labelcolor or fc)

for p in positions.keys():
    draw_arrow(p)

# test data
all_on_max = dict([(i, 1) for i in 'TCAG'] + \ 
                   [(i+j, 0.6) for i in 'TCAG' for j in 'TCAG'])
realistic_data = {
    'A':0.4,
    'T':0.3,
    'G':0.5,
    'C':0.2,
    'AT':0.4,
    'AC':0.3,
    'AG':0.2,
    'TA':0.2,
    'TC':0.3,
    'TG':0.4,
    'CT':0.2,
    'CG':0.3,
    'CA':0.2,
    'GA':0.1,
    'GT':0.4,
'GC':0.1,
}

extreme_data = {
    'A':0.75,
    'T':0.10,
    'G':0.10,
    'C':0.05,
    'AT':0.6,
    'AC':0.3,
    'AG':0.1,
    'TA':0.02,
    'TC':0.3,
    'TG':0.01,
    'CT':0.2,
    'CG':0.5,
    'CA':0.2,
    'GA':0.1,
    'GT':0.4,
    'GC':0.2,
}

sample_data = {
    'A':0.2137,
    'T':0.3541,
    'G':0.1946,
    'C':0.2376,
    'AT':0.0228,
    'AC':0.0684,
    'AG':0.2056,
    'TA':0.0315,
    'TC':0.0629,
    'TG':0.0315,
    'CT':0.1355,
    'CG':0.0401,
    'CA':0.0703,
    'GA':0.1824,
    'GT':0.0387,
    'GC':0.1106,
}

if __name__ == '__main__':
    from sys import argv
    d = None
    if len(argv) > 1:
        if argv[1] == 'full':
            d = all_on_max
            scaled = False
        elif argv[1] == 'extreme':
            d = extreme_data
            scaled = False
        elif argv[1] == 'realistic':
d = realistic_data
scaled = False
elif argv[1] == 'sample':
    d = sample_data
    scaled = True
if d is None:
    d = all_on_max
    scaled=False
if len(argv) > 2:
    display = argv[2]
else:
    display = 'length'

size = 4
figure(figsize=(size,size))

make_arrow_plot(d, display=display, linewidth=0.001, edgecolor=None,
                normalize_data=scaled, head_starts_at_zero=True, size=size)

draw()

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt

ax = plt.axes()
ax.arrow(0, 0, 0.5, 0.5, head_width=0.05, head_length=0.1, fc='k', ec='k')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_xscale("log")
ax1.set_yscale("log")
ax1.set_xlim(1e1, 1e3)
ax1.set_ylim(1e2, 1e3)
ax1.set_aspect(1)
ax1.set_title("adjustable = box")

ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_adjustable("datalim")
ax2.plot([1, 3, 10], [1, 9, 100], "o-")
ax2.set_xlim(1e-1, 1e2)
ax2.set_ylim(1e-1, 1e3)
ax2.set_aspect(1)
ax2.set_title("adjustable = datalim")

plt.draw()
plt.show()
#!/usr/bin/env python

from pylab import *

# create some data to use for the plot
dt = 0.001
r = exp(-t[:1000]/0.05)  # impulse response
x = randn(len(t))
s = convolve(x,r)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plot(t, s)
axis([0, 1, 1.1*amin(s), 2*amax(s)])
xlabel('time (s)')
ylabel('current (nA)')
title('Gaussian colored noise')

# this is an inset axes over the main axes
a = axes([.65, .6, .2, .2], axisbg='y')
n, bins, patches = hist(s, 400, normed=1)
title('Probability')
setp(a, xticks=[], yticks=[])  

# this is another inset axes over the main axes
a = axes([0.2, 0.6, .2, .2], axisbg='y')
plot(t[:len(r)], r)
title('Impulse response')
setp(a, xlim=(0,.2), xticks=[], yticks=[])  

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.15  pylab_examples example code: axes_props.py

#!/usr/bin/env python

"""
You can control the axis tick and grid properties
"""
from pylab import *

t = arange(0.0, 2.0, 0.01)
s = sin(2*pi*t)
plot(t, s)
grid(True)

# MATLAB style
xticklines = getp(gca(), 'xticklines')
yticklines = getp(gca(), 'yticklines')
xgridlines = getp(gca(), 'xgridlines')
ygridlines = getp(gca(), 'ygridlines')
xticklabels = getp(gca(), 'xticklabels')
yticklabels = getp(gca(), 'yticklabels')

setp(xticklines, 'linewidth', 3)
setp(yticklines, 'linewidth', 3)
setp(xgridlines, 'linestyle', '-')
setp(ygridlines, 'linestyle', '-')
setp(yticklabels, 'color', 'r', fontsize='medium')
setp(xticklabels, 'color', 'r', fontsize='medium')

show()

# the same script, python style
from pylab import *

t = arange(0.0, 2.0, 0.01)
s = sin(2*pi*t)
fig, ax = plt.subplots() 
ax.plot(t, s)
ax.grid(True)

ticklines = ax.get_xticklines()
ticklines.extend( ax.get_yticklines() )
gridlines = ax.get_xgridlines()
gridlines.extend( ax.get_ygridlines() )
ticklabels = ax.get_xticklabels()
ticklabels.extend( ax.get_yticklabels() )

for line in ticklines:
    line.set_linewidth(3)

for line in gridlines:
    line.set_linestyle('-')

for label in ticklabels:
    label.set_color('r')
    label.set_fontsize('medium')
show()

****

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 84.16 pylab_examples example code: axes_zoom_effect.py

```python
from matplotlib.transforms import Bbox, TransformedBbox, 
    blended_transform_factory
from mpl_toolkits.axes_grid1.inset_locator import BboxPatch, BboxConnector,
    BboxConnectorPatch

def connect_bbox(bbox1, bbox2,
    loc1a, loc2a, loc1b, loc2b,
    prop_lines, prop_patches=None):
    if prop_patches is None:
```
prop_patches = prop_lines.copy()
prop_patches["alpha"] = prop_patches.get("alpha", 1)*0.2

c1 = BboxConnector(bbox1, bbox2, loc1=loc1a, loc2=loc2a, **prop_lines)
c1.set_clip_on(False)
c2 = BboxConnector(bbox1, bbox2, loc1=loc1b, loc2=loc2b, **prop_lines)
c2.set_clip_on(False)

bbox_patch1 = BboxPatch(bbox1, **prop_patches)
bbox_patch2 = BboxPatch(bbox2, **prop_patches)

p = BboxConnectorPatch(bbox1, bbox2,
    #loc1a=3, loc2a=2, loc1b=4, loc2b=1,
    loc1a=loc1a, loc2a=loc2a, loc1b=loc1b, loc2b=loc2b,
    **prop_patches)
p.set_clip_on(False)

return c1, c2, bbox_patch1, bbox_patch2, p

def zoom_effect01(ax1, ax2, xmin, xmax, **kwargs):
    """
    ax1 : the main axes
    ax2 : the zoomed axes
    (xmin, xmax) : the limits of the colored area in both plot axes.

    connect ax1 & ax2. The x-range of (xmin, xmax) in both axes will
    be marked. The keywords parameters will be used to create
    patches.
    """

    trans1 = blended_transform_factory(ax1.transData, ax1.transAxes)
    trans2 = blended_transform_factory(ax2.transData, ax2.transAxes)

    bbox = Bbox.from_extents(xmin, 0, xmax, 1)

    mybbox1 = TransformedBbox(bbox, trans1)
    mybbox2 = TransformedBbox(bbox, trans2)

    prop_patches=kwargs.copy()
    prop_patches["ec"]="none"
    prop_patches["alpha"]=0.2

    c1, c2, bbox_patch1, bbox_patch2, p = \
    connect_bbox(mybbox1, mybbox2,
        loc1a=3, loc2a=2, loc1b=4, loc2b=1,
        prop_lines=kwargs, prop_patches=prop_patches)

    ax1.add_patch(bbox_patch1)
    ax2.add_patch(bbox_patch2)
    ax2.add_patch(c1)
    ax2.add_patch(c2)
import matplotlib.pyplot as plt

plt.figure(1, figsize=(5,5))
ax1 = plt.subplot(221)
ax2 = plt.subplot(212)
ax2.set_xlim(0, 1)
ax2.set_xlim(0, 5)
zoom_effect01(ax1, ax2, 0.2, 0.8)

ax1 = plt.subplot(222)
ax1.set_xlim(2, 3)
ax2.set_xlim(0, 5)
zoom_effect02(ax1, ax2)
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(-1, 2, 0.01)
s = np.sin(2 * np.pi * t)

plt.plot(t, s)

# draw a thick red hline at y=0 that spans the xrange
l = plt.axhline(linewidth=4, color='r')

# draw a default hline at y=1 that spans the xrange
l = plt.axhline(y=1)

# draw a default vline at x=1 that spans the yrange
l = plt.axvline(x=1)
# draw a thick blue vline at x=0 that spans the upper quadrant of
# the yrange
l = plt.axvline(x=0, ymin=0.75, linewidth=4, color='b')

# draw a default hline at y=.5 that spans the the middle half of
# the axes
l = plt.axhline(y=.5, xmin=0.25, xmax=0.75)

p = plt.axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)

p = plt.axvspan(1.25, 1.55, facecolor='g', alpha=0.5)

plt.axis([-1, 2, -1, 2])

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.18 pylab_examples example code: axis_equal_demo.py
This example is only interesting when run in interactive mode.

```python
from pylab import *

# Plot circle or radius 3

an = linspace(0,2*pi,100)

subplot(221)
plot( 3*cos(an), 3*sin(an) )
title('not equal, looks like ellipse',fontsize=10)

subplot(222)
plot( 3*cos(an), 3*sin(an) )
axis('equal')
title('equal, looks like circle',fontsize=10)

subplot(223)
plot( 3*cos(an), 3*sin(an) )
axis('equal')
axis([-3,3,-3,3])
title('looks like circle, even after changing limits',fontsize=10)

subplot(224)
plot( 3*cos(an), 3*sin(an) )
axis('equal')
axis([-3,3,-3,3])
plot([0,4],[0,4])
title('still equal after adding line',fontsize=10)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see `Search examples`)
# A stacked bar plot with errorbars

```python
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt

N = 5
menMeans = (20, 35, 30, 35, 27)
womenMeans = (25, 32, 34, 20, 25)
menStd = (2, 3, 4, 1, 2)
womenStd = (3, 5, 2, 3, 3)

ind = np.arange(N)  # the x locations for the groups
width = 0.35  # the width of the bars: can also be len(x) sequence

p1 = plt.bar(ind, menMeans, width, color='r', yerr=womenStd)
p2 = plt.bar(ind, womenMeans, width, color='y',
              bottom=menMeans, yerr=menStd)

plt.ylabel('Scores')
plt.title('Scores by group and gender')
plt.xticks(ind+width/2., ('G1', 'G2', 'G3', 'G4', 'G5'))
plt.yticks(np.arange(0, 81, 10))
```

Scores by group and gender

Men

Women
plt.legend( (p1[0], p2[0]), ('Men', 'Women') )
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.20  pylab_examples example code: barb_demo.py
Demonstration of wind barb plots

import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5, 5, 5)
X, Y = np.meshgrid(x, x)
U, V = 12*X, 12*Y

data = 
[-1.5, .5, -6, -6],
(1, -1, -46, 46),
(-3, -1, 11, -11),
(1, 1.5, 80, 80),
(0.5, 0.25, 25, 15),
(-1.5, -0.5, -5, 40]

data = np.array(data, dtype=[('x', np.float32), ('y', np.float32), ('u', np.float32), ('v', np.float32)])

#Default parameters, uniform grid
ax = plt.subplot(2, 2, 1)
ax.barbs(X, Y, U, V)

#Arbitrary set of vectors, make them longer and change the pivot point
#(point around which they're rotated) to be the middle
ax = plt.subplot(2,2,2)
ax.barbs(data['x'], data['y'], data['u'], data['v'], length=8, pivot='middle')

#Showing colormapping with uniform grid. Fill the circle for an empty barb,
don't round the values, and change some of the size parameters
ax = plt.subplot(2,2,3)
ax.barbs(X, Y, U, V, np.sqrt(U*U + V*V), fill_empty=True, rounding=False,
sizes=dict(emptybarb=0.25, spacing=0.2, height=0.3))

#Change colors as well as the increments for parts of the barbs
ax = plt.subplot(2,2,4)
ax.barbs(data['x'], data['y'], data['u'], data['v'], flagcolor='r',
barbcolor=['b','g'], barb_increments=dict(half=10, full=20, flag=100),
flip_barb=True)

#Masked arrays are also supported
masked_u = np.ma.masked_array(data['u'])
masked_u[4] = 1000 #Bad value that should not be plotted when masked
masked_u[4] = np.ma.masked

#Identical plot to panel 2 in the first figure, but with the point at
#(0.5, 0.25) missing (masked)
fig2 = plt.figure()
ax = fig2.add_subplot(1, 1, 1)
ax.barbs(data['x'], data['y'], masked_u, data['v'], length=8, pivot='middle')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Bar chart demo with pairs of bars grouped for easy comparison.

```python
import numpy as np
import matplotlib.pyplot as plt

n_groups = 5
means_men = (20, 35, 30, 35, 27)
std_men = (2, 3, 4, 1, 2)
means_women = (25, 32, 34, 20, 25)
std_women = (3, 5, 2, 3, 3)

fig, ax = plt.subplots()
index = np.arange(n_groups)
bar_width = 0.35
opacity = 0.4
error_config = {'ecolor': '0.3'}
```

Scores by group and gender

Scores by group and gender

Men

Women
rects1 = plt.bar(index, means_men, bar_width,
    alpha=opacity,
    color='b',
    yerr=std_men,
    error_kw=error_config,
    label='Men')

rects2 = plt.bar(index + bar_width, means_women, bar_width,
    alpha=opacity,
    color='r',
    yerr=std_women,
    error_kw=error_config,
    label='Women')

plt.xlabel('Group')
plt.ylabel('Scores')
plt.title('Scores by group and gender')
plt.xticks(index + bar_width, ('A', 'B', 'C', 'D', 'E'))
plt.legend()

plt.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Thanks Josh Hemann for the example

This example comes from an application in which grade school gym teachers wanted to be able to show parents how their child did across a handful of fitness tests, and importantly, relative to how other children did. To extract the plotting code for demo purposes, we'll just make up some data for little Johnny Doe...

import numpy as np
import matplotlib.pyplot as plt
import pylab
from matplotlib.ticker import MaxNLocator

student = 'Johnny Doe'
grade = 2
gender = 'boy'
cohortSize = 62  # The number of other 2nd grade boys
numTests = 5
testNames = ['Pacer Test', 'Flexed Arm\n Hang', 'Mile Run', 'Agility', 'Push Ups']
testMeta = ['laps', 'sec', 'min:sec', 'sec', '']
scores = [7, '48', '12:52', '17', '14']
rankings = np.round(np.random.uniform(0, 1, numTests)*100, 0)

fig, ax1 = plt.subplots(figsize=(9, 7))
plt.subplots_adjust(left=0.115, right=0.88)
fig.canvas.set_window_title('Eldorado K-8 Fitness Chart')
pos = np.arange(numTests)+0.5  # Center bars on the Y-axis ticks
rects = ax1.barh(pos, rankings, align='center', height=0.5, color='m')

ax1.axis([0, 100, 0, 5])
pylab.yticks(pos, testNames)
ax1.set_title('Johnny Doe')
plt.text(50, -0.5, 'Cohort Size: ' + str(cohortSize), horizontalalignment='center', size='small')

# Set the right-hand Y-axis ticks and labels and set X-axis tick marks at the deciles
ax2 = ax1.twinx()
ax2.plot([100, 100], [0, 0.5], 'white', alpha=0.1)
ax2.set_major_locator(MaxNLocator(11))
xticks = pylab.setp(ax2, xticklabels=['0', '10', '20', '30', '40', '50', '60', '70', '80', '90', '100'])
ax2.set_yticks(pos)
ax2.set_yticklabels(scoreLabels)
ax2.set_ylim(ax1.get_ylim())

# Build up the score labels for the right Y-axis by first appending a carriage return to each string and then tacking on the appropriate meta information (i.e., 'laps' vs 'seconds'). We want the labels centered on the ticks, so if there is no meta info (like for pushups), then don't add the carriage return to the string

def withnew(i, scr):
    if testMeta[i] != '':
        return '%s
' % scr
    else:
        return scr

scoreLabels = [withnew(i, scr) for i, scr in enumerate(scores)]
scoreLabels = [i+j for i, j in zip(scoreLabels, testMeta)]

# set the tick locations
ax2.set_yticks(pos)
# set the tick labels
ax2.set_yticklabels(scoreLabels)
# make sure that the limits are set equally on both yaxis so the ticks line up
ax2.set_ylim(ax1.get_ylim())
ax2.set_ylabel('Test Scores')

# Make list of numerical suffixes corresponding to position in a list
suffixes = ['th', 'st', 'nd', 'rd', 'th', 'th', 'th', 'th', 'th', 'th']

ax2.set_xlabel('Percentile Ranking Across ' + str(grade) + suffixes[grade] + ' Grade ' + gender.title() + 's')

# Lastly, write in the ranking inside each bar to aid in interpretation
for rect in rects:
    # Rectangle widths are already integer-valued but are floating
    # type, so it helps to remove the trailing decimal point and 0 by
    # converting width to int type
    width = int(rect.get_width())

    # Figure out what the last digit (width modulo 10) so we can add
    # the appropriate numerical suffix (e.g., 1st, 2nd, 3rd, etc)
    lastDigit = width % 10
    # Note that 11, 12, and 13 are special cases
    if (width == 11) or (width == 12) or (width == 13):
        suffix = 'th'
    else:
        suffix = suffixes[lastDigit]

    rankStr = str(width) + suffix
    if (width < 5):
        xloc = width + 1  # Shift the text to the right side of the right edge
        clr = 'black'  # Black against white background
        align = 'left'
    else:
        xloc = 0.98*width  # Shift the text to the left side of the right edge
        clr = 'white'  # White on magenta
        align = 'right'

    # Center the text vertically in the bar
    yloc = rect.get_y()+rect.get_height()/2.0
    ax1.text(xloc, yloc, rankStr, horizontalalignment=align, verticalalignment='center', color=clr, weight='bold')

plt.show()
from matplotlib.pyplot import figure, show, cm
from numpy import where
from numpy.random import rand

# the bar
x = where(rand(500)>0.7, 1.0, 0.0)

axprops = dict(xticks=[], yticks=[])  
barprops = dict(aspect='auto', cmap=cm.binary, interpolation='nearest')

fig = figure()

# a vertical barcode -- this is broken at present
x.shape = len(x), 1
ax = fig.add_axes([0.1, 0.3, 0.1, 0.6], **axprops)
ax.imshow(x, **barprops)

x = x.copy()

# a horizontal barcode
x.shape = 1, len(x)
ax = fig.add_axes([0.3, 0.1, 0.6, 0.1], **axprops)
ax.imshow(x, **barprops)
show()

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

**84.24 `pylab_examples` example code: boxplot_demo.py**

![Boxplot Diagram](image-url)
#!/usr/bin/python

# Example boxplot code

from pylab import *

# fake up some data
spread = rand(50) * 100
center = ones(25) * 50
flier_high = rand(10) * 100 + 100
flier_low = rand(10) * -100
data = concatenate((spread, center, flier_high, flier_low), 0)

# basic plot
boxplot(data)

# notched plot
figure()
boxplot(data, 1)

# change outlier point symbols
figure()
boxplot(data, 0, 'gD')
# don't show outlier points
figure()
boxplot(data,0,'')

# horizontal boxes
figure()
boxplot(data,0,'rs',0)

# change whisker length
figure()
boxplot(data,0,'rs',0,0.75)

# fake up some more data
spread= rand(50) * 100
center = ones(25) * 40
flier_high = rand(10) * 100 + 100
flier_low = rand(10) * -100
d2 = concatenate( (spread, center, flier_high, flier_low), 0 )
data.shape = (-1, 1)
d2.shape = (-1, 1)
#data = concatenate( (data, d2), 1 )
# Making a 2-D array only works if all the columns are the
# same length. If they are not, then use a list instead.
# This is actually more efficient because boxplot converts
# a 2-D array into a list of vectors internally anyway.
data = [data, d2, d2[::2,0]]
# multiple box plots on one figure
figure()
boxplot(data)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Comparison of IID Bootstrap Resampling Across Five Distributions

Value

0.99 1.0 2.68 2.55 0.69 0.7 7.98 6.31 7.55 7.65

Distribution

Normal(1,1) Normal(1,1) Lognormal(1,1) Lognormal(1,1) Exp(1) Exp(1) Gumbel(6,4) Gumbel(6,4) Triangular(2,9,11) Triangular(2,9,11)

500 Random Numbers
IID Bootstrap Resample
Average Value

""
Thanks Josh Hemann for the example
""

# Generate some data from five different probability distributions, # each with different characteristics. We want to play with how an IID # bootstrap resample of the data preserves the distributional # properties of the original sample, and a boxplot is one visual tool # to make this assessment

numDists = 5
randomDists = ['Normal(1,1)', 'Lognormal(1,1)', 'Exp(1)', 'Gumbel(6,4)', 'Triangular(2,9,11)']

N = 500
norm = np.random.normal(1, 1, N)
logn = np.random.lognormal(1, 1, N)
expo = np.random.exponential(1, N)
gumb = np.random.gumbel(6, 4, N)
tria = np.random.triangular(2, 9, 11, N)

# Generate some random indices that we'll use to resample the original data # arrays. For code brevity, just use the same random indices for each array

""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

# Generate some data from five different probability distributions, # each with different characteristics. We want to play with how an IID # bootstrap resample of the data preserves the distributional # properties of the original sample, and a boxplot is one visual tool # to make this assessment

numDists = 5
randomDists = ['Normal(1,1)', 'Lognormal(1,1)', 'Exp(1)', 'Gumbel(6,4)', 'Triangular(2,9,11)']

N = 500
norm = np.random.normal(1, 1, N)
logn = np.random.lognormal(1, 1, N)
expo = np.random.exponential(1, N)
gumb = np.random.gumbel(6, 4, N)
tria = np.random.triangular(2, 9, 11, N)

# Generate some random indices that we'll use to resample the original data # arrays. For code brevity, just use the same random indices for each array

""
bootstrapIndices = np.random.random_integers(0, N-1, N)
normBoot = norm[bootstrapIndices]
expoBoot = expo[bootstrapIndices]
gumbBoot = gumb[bootstrapIndices]
lognBoot = logn[bootstrapIndices]
triaboot = tria[bootstrapIndices]
data = [norm, normBoot, logn, lognBoot, expo, expoBoot, gumb, gumbBoot, tria, triaBoot]

fig, ax1 = plt.subplots(figsize=(10,6))
fig.canvas.set_window_title('A Boxplot Example')
plt.subplots_adjust(left=0.075, right=0.95, top=0.9, bottom=0.25)

bp = plt.boxplot(data, notch=0, sym='+', vert=1, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')
plt.setp(bp['fliers'], color='red', marker= '+')

# Add a horizontal grid to the plot, but make it very light in color
# so we can use it for reading data values but not be distracting
ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey', alpha=0.5)
ax1.set_axisbelow(True)
ax1.set_title('Comparison of IID Bootstrap Resampling Across Five Distributions')
ax1.set_xlabel('Distribution')
ax1.set_ylabel('Value')

# Now fill the boxes with desired colors
boxColors = ['darkkhaki', 'royalblue']
numBoxes = numDists * 2
medians = range(numBoxes)
for i in range(numBoxes):
    box = bp['boxes'][i]
    boxX = []
    boxY = []
    for j in range(5):
        boxX.append(box.get_xdata()[j])
        boxY.append(box.get_ydata()[j])
    boxCoords = zip(boxX, boxY)
    # Alternate between Dark Khaki and Royal Blue
    k = i % 2
    boxPolygon = Polygon(boxCoords, facecolor=boxColors[k])
    ax1.add_patch(boxPolygon)
    # Now draw the median lines back over what we just filled in
    med = bp['medians'][i]
    medianX = []
    medianY = []
    for j in range(2):
        medianX.append(med.get_xdata()[j])
        medianY.append(med.get_ydata()[j])
plt.plot(medianX, medianY, 'k')
medians[i] = medianY[0]

# Finally, overplot the sample averages, with horizontal alignment
# in the center of each box
plt.plot([np.average(med.get_xdata())], [np.average(data[i])],
color='w', marker='*', markeredgewidth=0)

# Set the axes ranges and axes labels
ax1.set_xlim(0.5, numBoxes+0.5)
top = 40
bottom = -5
ax1.set_ylim(bottom, top)
xtickNames = plt.setp(ax1, xticklabels=np.repeat(randomDists, 2))
plt.setp(xtickNames, rotation=45, fontsize=8)

# Due to the Y-axis scale being different across samples, it can be
# hard to compare differences in medians across the samples. Add upper
# X-axis tick labels with the sample medians to aid in comparison
# (just use two decimal places of precision)
pos = np.arange(numBoxes)+1
upperLabels = [str(np.round(s, 2)) for s in medians]
weights = ['bold', 'semibold']
for tick,label in zip(range(numBoxes),ax1.get_xticklabels()):
    k = tick % 2
    ax1.text(pos[tick], top-(top*0.05), upperLabels[tick],
              horizontalalignment='center', size='x-small', weight=weights[k],
              color=boxColors[k])

# Finally, add a basic legend
plt.figtext(0.80, 0.08, str(N) + ' Random Numbers',
            backgroundcolor=boxColors[0], color='black', weight='roman',
            size='x-small')
plt.figtext(0.80, 0.045, 'IID Bootstrap Resample',
            backgroundcolor=boxColors[1],
            color='white', weight='roman', size='x-small')
plt.figtext(0.80, 0.015, '*', color='white', backgroundcolor='silver',
            weight='roman', size='medium')
plt.figtext(0.815, 0.013, 'Average Value', color='black', weight='roman',
            size='x-small')

plt.show()
import matplotlib.pyplot as plt
import numpy as np

def fakeBootStrapper(n):
    '''
    This is just a placeholder for the user's method of
    bootstrapping the median and its confidence intervals.
    
    Returns an arbitrary median and confidence intervals
    packed into a tuple
    '''
    if n == 1:
        med = 0.1
        CI = (-0.25, 0.25)
    else:
        med = 0.2
        CI = (-0.35, 0.50)
    return med, CI
np.random.seed(2)
inc = 0.1
e1 = np.random.normal(0, 1, size=(500,))
e2 = np.random.normal(0, 1, size=(500,))
e3 = np.random.normal(0, 1 + inc, size=(500,))
e4 = np.random.normal(0, 1 + 2*inc, size=(500,))

treatments = [e1, e2, e3, e4]
med1, CI1 = fakeBootStrapper(1)
med2, CI2 = fakeBootStrapper(2)
medians = [None, None, med1, med2]
conf_intervals = [None, None, CI1, CI2]

fig, ax = plt.subplots()
pos = np.array(range(len(treatments)))+1
bp = ax.boxplot(treatments, sym='k+', positions=pos,
                notch=1, bootstrap=5000,
                usermedians=medians,
                conf_intervals=conf_intervals)

ax.set_xlabel('treatment')
ax.set_ylabel('response')
plt.setp(bp['whiskers'], color='k', linestyle='-' )
plt.setp(bp['fliers'], markersize=3.0)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.27 `pylab_examples` example code: `break.py`

Distance Histograms by Category

```python
#!/usr/bin/env python
from pylab import *

gcf().text(0.5, 0.95, 'Distance Histograms by Category is a really long title')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
Broken axis example, where the y-axis will have a portion cut out.

```python
import matplotlib.pylab as plt
import numpy as np

pts = np.array([0.015, 0.166, 0.133, 0.159, 0.041, 0.024, 0.195, 0.039, 0.161, 0.018, 0.143, 0.056, 0.125, 0.096, 0.094, 0.051, 0.043, 0.021, 0.138, 0.075, 0.109, 0.195, 0.05, 0.074, 0.079, 0.155, 0.02, 0.01, 0.061, 0.008])

# Now let's make two outlier points which are far away from everything.
pts[[3, 14]] += .8

# If we were to simply plot pts, we'd lose most of the interesting
details due to the outliers. So let's 'break' or 'cut-out' the y-axis
# into two portions - use the top (ax) for the outliers, and the bottom
# (ax2) for the details of the majority of our data
f, (ax, ax2) = plt.subplots(2, 1, sharex=True)
```
# plot the same data on both axes
ax.plot(pts)
ax2.plot(pts)

# zoom-in / limit the view to different portions of the data
ax.set_ylim(.78, 1.) # outliers only
ax2.set_ylim(0, .22) # most of the data

# hide the spines between ax and ax2
ax.spines['bottom'].set_visible(False)
ax2.spines['top'].set_visible(False)
ax.xaxis.tick_top()
ax.tick_params(labeltop='off') # don't put tick labels at the top
ax2.xaxis.tick_bottom()

# This looks pretty good, and was fairly painless, but you can get that
# cut-out diagonal lines look with just a bit more work. The important
# thing to know here is that in axes coordinates, which are always
# between 0-1, spine endpoints are at these locations (0,0), (0,1),
# (1,0), and (1,1). Thus, we just need to put the diagonals in the
# appropriate corners of each of our axes, and so long as we use the
# right transform and disable clipping.

d = .015 # how big to make the diagonal lines in axes coordinates
# arguments to pass plot, just so we don't keep repeating them
kwargs = dict(transform=ax.transAxes, color='k', clip_on=False)
ax.plot((-d,+d),(-d,+d), **kwargs) # top-left diagonal
ax.plot((1-d,1+d),(-d,+d), **kwargs) # top-right diagonal
kwargs.update(transform=ax2.transAxes) # switch to the bottom axes
ax2.plot((-d,+d),(1-d,1+d), **kwargs) # bottom-left diagonal
ax2.plot((1-d,1+d),(1-d,1+d), **kwargs) # bottom-right diagonal

# What's cool about this is that now if we vary the distance between
# ax and ax2 via f.subplots_adjust(hspace=...) or plt.subplot_tool(),
# the diagonal lines will move accordingly, and stay right at the tips
# of the spines they are 'breaking'

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Make a "broken" horizontal bar plot, i.e., one with gaps

```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.broken_barh([(110, 30), (150, 10)], (10, 9), facecolors='blue')
ax.broken_barh([(10, 50), (100, 20), (130, 10)], (20, 9),
               facecolors=('red', 'yellow', 'green'))
ax.set_ylim(5, 35)
ax.set_xlim(0, 200)
ax.set_xlabel('seconds since start')
ax.set_yticks([15, 25])
ax.set_yticklabels(['Bill', 'Jim'])
ax.grid(True)
ax.annotate('race interrupted', (61, 25), xytext=(.8, .9), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=16, horizontalalignment='right', verticalalignment='top')
plt.show()
```
84.30  *pylab_examples* example code: centered_ticklabels.py

# sometimes it is nice to have ticklabels centered.  *mpl* currently
# associates a label with a tick, and the label can be aligned
# 'center', 'left', or 'right' using the horizontal alignment property:
#
# for label in ax.xaxis.get_xticklabels():
#    label.set_horizontalalignment('right')
#
# but this doesn't help center the label between ticks.  One solution
# is to "face it".  Use the minor ticks to place a tick centered
# between the major ticks.  Here is an example that labels the months,
# centered between the ticks

import numpy as np
import matplotlib.cbook as cbook
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt
# load some financial data; apple's stock price
fh = cbook.get_sample_data('aapl.npy.gz')
r = np.load(fh); fh.close()
r = r[-250:]  # get the last 250 days

fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close)

ax.xaxis.set_major_locator(dates.MonthLocator())
ax.xaxis.set_minor_locator(dates.MonthLocator(bymonthday=15))

ax.xaxis.set_major_formatter(ticker.NullFormatter())
ax.xaxis.set_minor_formatter(dates.DateFormatter('%b'))

for tick in ax.xaxis.get_minor_ticks():
    tick.tick1line.set_markersize(0)
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

imid = len(r)/2
ax.set_xlabel(str(r.date[imid].year))
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)
Clip a line according to the current xlimits, and change the marker style when zoomed in.

It is not clear this example is still needed or valid; clipping is now automatic for Line2D objects when x is sorted in ascending order.

```python
from matplotlib.lines import Line2D
import numpy as np
from pylab import figure, show

class ClippedLine(Line2D):
    ""
    Clip the xlimits to the axes view limits -- this example assumes x is sorted ""
    
def __init__(self, ax, *args, **kwargs):
        Line2D.__init__(self, *args, **kwargs)
        self.ax = ax
```

.. code-block:: python

    def set_data(self, *args, **kwargs):
        Line2D.set_data(self, *args, **kwargs)
        self.recache()
        self.xorig = np.array(self._x)
        self.yorig = np.array(self._y)

    def draw(self, renderer):
        xlim = self.ax.get_xlim()
        ind0, ind1 = np.searchsorted(self.xorig, xlim)
        self._x = self.xorig[ind0:ind1]
        self._y = self.yorig[ind0:ind1]
        N = len(self._x)
        if N<1000:
            self._marker = 's'
            self._linestyle = '-'
        else:
            self._marker = None
            self._linestyle = '-'

        Line2D.draw(self, renderer)

fig = figure()
ax = fig.add_subplot(111, autoscale_on=False)

t = np.arange(0.0, 100.0, 0.01)
s = np.sin(2*np.pi*t)
line = ClippedLine(ax, t, s, color='g', ls='-', lw=2)
ax.add_line(line)
ax.set_xlim(10,30)
ax.set_ylim(-1.1,1.1)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

""" Compute the coherence of two signals """

import numpy as np
import matplotlib.pyplot as plt

# make a little extra space between the subplots
plt.subplots_adjust(wspace=0.5)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))                  # white noise 1
nse2 = np.random.randn(len(t))                  # white noise 2
r = np.exp(-t/0.05)

cnse1 = np.convolve(nse1, r, mode='same')*dt   # colored noise 1
   # colored noise 2

s1 = 0.01*np.sin(2*np.pi*10*t) + cnse1
s2 = 0.01*np.sin(2*np.pi*10*t) + cnse2
```python
plt.subplot(211)
plt.plot(t, s1, 'b-', t, s2, 'g-')
plt.xlim(0, 5)
plt.xlabel('time')
plt.ylabel('s1 and s2')
plt.grid(True)

plt.subplot(212)
cxy, f = plt.cohere(s1, s2, 256, 1./dt)
plt.ylabel('coherence')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.33 `pylab_examples` example code: `color_by_yvalue.py`

```python
# use masked arrays to plot a line with different colors by y-value
from numpy import logical_or, arange, sin, pi
from numpy import ma
from matplotlib.pyplot import plot, show

t = arange(0.0, 2.0, 0.01)
```
s = sin(2*pi*t)

upper = 0.77
lower = -0.77

supper = ma.masked_where(s < upper, s)
slower = ma.masked_where(s > lower, s)
smiddle = ma.masked_where(logical_or(s<lower, s>upper), s)

plot(t, slower, 'r', t, smiddle, 'b', t, supper, 'g')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.34 pylab_examples example code: color_demo.py

#!/usr/bin/env python
"""
matplotlib gives you 4 ways to specify colors,

1) as a single letter string, ala MATLAB
"""
2) as an html style hex string or html color name

3) as an R,G,B tuple, where R,G,B, range from 0-1

4) as a string representing a floating point number from 0 to 1, corresponding to shades of gray.

See help(colors) for more info.

```
from pylab import *

subplot(111, axisbg='darkslategray')
#subplot(111, axisbg='#ababab')
t = arange(0.0, 2.0, 0.01)
s = sin(2*pi*t)
plot(t, s, 'y')
xlabel('time (s)', color='r')
ylabel('voltage (mV)', color='0.5') # grayscale color
title('About as silly as it gets, folks', color='#afeeee')
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Gaussian noise with vertical colorbar
Produce custom labelling for a colorbar.

Contributed by Scott Sinclair

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from numpy.random import randn

# Make plot with vertical (default) colorbar
fig, ax = plt.subplots()
data = np.clip(randn(250, 250), -1, 1)
cax = ax.imshow(data, interpolation='nearest', cmap=cm.coolwarm)
ax.set_title('Gaussian noise with vertical colorbar')

cbar = fig.colorbar(cax, ticks=[-1, 0, 1])
cbar.ax.set_yticklabels(['< -1', '0', '> 1'])

# Make plot with horizontal colorbar
fig, ax = plt.subplots()```
```python
data = np.clip(randn(250, 250), -1, 1)
cax = ax.imshow(data, interpolation='nearest', cmap=cm.afmhot)
ax.set_title('Gaussian noise with horizontal colorbar')
cbar = fig.colorbar(cax, ticks=[-1, 0, 1], orientation='horizontal')
cbar.ax.set_xticklabels(['Low', 'Medium', 'High'])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 84.36 `pylab_examples` example code: `colours.py`

[source code]

```python
#!/usr/bin/env python
# -*- noplot -*-

""
Some simple functions to generate colours.
""
import numpy as np
from matplotlib.colors import colorConverter

def pastel(colour, weight=2.4):
    """ Convert colour into a nice pastel shade""
    rgb = np.asarray(colorConverter.to_rgb(colour))
    # scale colour
    maxc = max(rgb)
    if maxc < 1.0 and maxc > 0:
        # scale colour
        scale = 1.0 / maxc
        rgb = rgb * scale
    # now decrease saturation
    total = rgb.sum()
    slack = 0
    for x in rgb:
        slack += 1.0 - x
        # want to increase weight from total to weight
        # pick x s.t. slack * x == weight - total
        # x = (weight - total) / slack
        x = (weight - total) / slack
    rgb = [c + (x * (1.0-c)) for c in rgb]
    return rgb

def get_colours(n):
    """ Return n pastel colours. ""
    base = np.asarray([[1,0,0], [0,1,0], [0,0,1]])
```

84.36. `pylab_examples` example code: `colours.py` 1993
if n <= 3:
    return base[0:n]

# how many new colours to we need to insert between
# red and green and between green and blue?
needed = (((n - 3) + 1) / 2, (n - 3) / 2)

colours = []
for start in (0, 1):
    for x in np.linspace(0, 1, needed[start]+2):
        colours.append((base[start] * (1.0 - x)) +
                        (base[start+1] * x))

return [pastel(c) for c in colours[0:n]]

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.37 pylab_examples example code: contour_demo.py
labels at selected locations

The image shows a contour plot with labels at selected locations. The plot has contours at various levels, including 0.000, 0.500, 1.000, 1.500, and -0.500. The axes range from -3 to 3 on the x-axis and from -2 to 2 on the y-axis.
Single color - negative contours dashed
Crazy lines
#!/usr/bin/env python

"""
Illustrate simple contour plotting, contours on an image with a colorbar for the contours, and labelled contours.

See also contour_image.py.
"""

import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)
# Create a simple contour plot with labels using default colors. The
# inline argument to clabel will control whether the labels are draw
# over the line segments of the contour, removing the lines beneath
# the label
plt.figure()
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('Simplest default with labels')

# contour labels can be placed manually by providing list of positions
# (in data coordinate). See ginput_manual_clabel.py for interactive
# placement.
plt.figure()
CS = plt.contour(X, Y, Z)
manual_locations = [(-1, -1.4), (-0.62, -0.7), (-2, 0.5), (1.7, 1.2), (2.0, 1.4), (2.4, 1.7)]
plt.clabel(CS, inline=1, fontsize=10, manual=manual_locations)
plt.title('labels at selected locations')

# You can force all the contours to be the same color.
plt.figure()
CS = plt.contour(X, Y, Z,
                 colors='k',  # negative contours will be dashed by default
                 )
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Single color - negative contours dashed')

# You can set negative contours to be solid instead of dashed:
matplotlib.rcParams['contour.negative_linestyle'] = 'solid'
plt.figure()
CS = plt.contour(X, Y, Z, 6,
                 colors='k',  # negative contours will be dashed by default
                 )
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Single color - negative contours solid')

# And you can manually specify the colors of the contour
plt.figure()
CS = plt.contour(X, Y, Z, 6,
                 linewidths=np.arange(.5, 4, .5),
                 colors=('r', 'green', 'blue', (1,1,0), '#af8eee', '0.5')
                 )
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Crazy lines')

# Or you can use a colormap to specify the colors; the default
# colormap will be used for the contour lines
plt.figure()
im = plt.imshow(Z, interpolation='bilinear', origin='lower',
                 )
```python
cmap=cm.gray, extent=(-3,3,-2,2))
levels = np.arange(-1.2, 1.6, 0.2)
CS = plt.contour(Z, levels,
    origin='lower',
    linewidths=2,
    extent=(-3,3,-2,2))

#Thicken the zero contour.
zc = CS.collections[6]
plt.setp(zc, linewidth=4)
plt.clabel(CS, levels[1::2],
    inline=1,
    fmt='%.1f',
    fontsize=14)

# make a colorbar for the contour lines
CB = plt.colorbar(CS, shrink=0.8, extend='both')

plt.title('Lines with colorbar')
# plt.hot() # Now change the colormap for the contour lines and colorbar
plt.flag()

# We can still add a colorbar for the image, too.
CBI = plt.colorbar(im, orientation='horizontal', shrink=0.8)

# This makes the original colorbar look a bit out of place,
# so let's improve its position.
1,b,w,h = plt.gca().get_position().bounds
ll,bb,ww,hh = CB.ax.get_position().bounds
CB.ax.set_position([ll, b+0.1*h, ww, h*0.8])

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
#!/usr/bin/env python

"""Test combinations of contouring, filled contouring, and image plotting. For contour labelling, see contour_demo.py.

The emphasis in this demo is on showing how to make contours register correctly on images, and on how to get both of them oriented as desired. In particular, note the usage of the "origin" and "extent" keyword arguments to imshow and contour."

from pylab import *

# Default delta is large because that makes it fast, and it illustrates # the correct registration between image and contours.

delta = 0.5
extent = (-3,4,-4,3)

x = arange(-3.0, 4.001, delta)
y = arange(-4.0, 3.001, delta)
X, Y = meshgrid(x, y)
Z1 = bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = (Z1 - Z2) * 10

levels = arange(-2.0, 1.601, 0.4)  # Boost the upper limit to avoid truncation errors.
norm = cm.colors.Normalize(vmax=abs(Z).max(), vmin=-abs(Z).max())
cmap = cm.PRGn

figure()

subplot(2,2,1)
cset1 = contourf(X, Y, Z, levels,
                   cmap=cm.get_cmap(cmap, len(levels)-1),
                   norm=norm,
                   )

# It is not necessary, but for the colormap, we need only the number of levels minus 1. To avoid discretization error, use either this number or a large number such as the default (256).

# If we want lines as well as filled regions, we need to call contour separately; don't try to change the edgecolor or edgewidth of the polygons in the collections returned by contour.
# Use levels output from previous call to guarantee they are the same.
cset2 = contour(X, Y, Z, cset1.levels,
                   colors = 'k',
                   hold='on')

# We don't really need dashed contour lines to indicate negative regions, so let's turn them off.
for c in cset2.collections:
    c.set_linestyle('solid')

# It is easier here to make a separate call to contour than to set up an array of colors and linewidths.
# We are making a thick green line as a zero contour.
# Specify the zero level as a tuple with only 0 in it.
cset3 = contour(X, Y, Z, (0,),
                   colors = 'g',
                   linewidhts = 2,
                   hold='on')
title('Filled contours')
colorbar(cset1)
#hot()

subplot(2,2,2)

imshow(Z, extent=extent, cmap=cmap, norm=norm)
v = axis()
contour(Z, levels, hold='on', colors = 'k',
        origin='upper', extent=extent)
axis(v)
title("Image, origin 'upper'")

subplot(2,2,3)

imshow(Z, origin='lower', extent=extent, cmap=cmap, norm=norm)
v = axis()
contour(Z, levels, hold='on', colors = 'k',
        origin='lower', extent=extent)
axis(v)
title("Image, origin 'lower'")

subplot(2,2,4)

# We will use the interpolation "nearest" here to show the actual
# image pixels.
# Note that the contour lines don't extend to the edge of the box.
# This is intentional. The Z values are defined at the center of each
# image pixel (each color block on the following subplot), so the
# domain that is contoured does not extend beyond these pixel centers.
im = imshow(Z, interpolation='nearest', extent=extent, cmap=cmap, norm=norm)
v = axis()
contour(Z, levels, hold='on', colors = 'k',
        origin='image', extent=extent)
axis(v)
ylim = get(gca(), 'ylim')
setp(gca(), ylim=ylim[::-1])
title("Image, origin from rc, reversed y-axis")
colorbar(im)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.39  `pylab_examples` example code: `contour_label_demo.py`
#!/usr/bin/env python

""
Illustrate some of the more advanced things that one can do with
contour labels.

See also contour_demo.py.
""

import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

# Define our surface
# Define our surface
delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)

# Make contour labels using creative float classes
# Follows suggestion of Manuel Metz
plt.figure()
# Basic contour plot
CS = plt.contour(X, Y, Z)

# Define a class that forces representation of float to look a certain way
# This remove trailing zero so '1.0' becomes '1'
class nf(float):
    def __repr__(self):
        str = '%.1f' % (self.__float__(),)
        if str[-1]=='0':
            return '%.0f' % self.__float__()
        else:
            return '%.1f' % self.__float__()

# Recast levels to new class
CS.levels = [nf(val) for val in CS.levels]

# Label levels with specially formatted floats
if plt.rcParams['text.usetex']:
    fmt = r'\%r'
else:
    fmt = '%r

plt.clabel(CS, CS.levels, inline=True, fmt=fmt, fontsize=10)

# Label contours with arbitrary strings using a
dictionary
plt.figure()
# Basic contour plot
CS = plt.contour(X, Y, Z)

fmt = {}
strs = [ 'first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh' ]
for l,s in zip( CS.levels, strs):
    fmt[l] = s

# Label every other level using strings
plt.clabel(CS,CS.levels[:2],inline=True,fmt=fmt,fontsize=10)

# Use a Formatter
plt.figure()
CS = plt.contour(X, Y, 100**Z, locator=plt.LogLocator())
fmt = ticker.LogFormatterMathtext()
fmt.create_dummy_axis()
plt.clabel(CS, CS.levels, fmt=fmt)
plt.title("$100^Z$")
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.40  pylab_examples example code: contourf_demo.py
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt

origin = 'lower'
#origin = 'upper'

delta = 0.025

x = y = np.arange(-3.0, 3.01, delta)
X, Y = np.meshgrid(x, y)
Z1 = plt.mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = plt.mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10 * (Z1 - Z2)

nr, nc = Z.shape

# put NaNs in one corner:
Z[-nr//6:, -nc//6:] = np.nan
# contourf will convert these to masked

Z = np.ma.array(Z)

# mask another corner:
Z[:nr//6, :nc//6] = np.ma.masked
# mask a circle in the middle:
interior = np.sqrt((X**2) + (Y**2)) < 0.5
Z[interior] = np.ma.masked

# We are using automatic selection of contour levels;
# this is usually not such a good idea, because they don't
# occur on nice boundaries, but we do it here for purposes
# of illustration.
CS = plt.contourf(X, Y, Z, 10, # [-1, -0.1, 0, 0.1],
                  # alpha=0.5,
                  cmap=plt.cm.bone,
                  origin=origin)

# Note that in the following, we explicitly pass in a subset of
# the contour levels used for the filled contours. Alternatively,
# We could pass in additional levels to provide extra resolution,
# or leave out the levels kwarg to use all of the original levels.
CS2 = plt.contour(CS, levels=CS.levels[:2],
                  colors = 'r',
                  origin=origin,
                  hold='on')

plt.title('Nonsense (3 masked regions)')
plt.xlabel('word length anomaly')
plt.ylabel('sentence length anomaly')

# Make a colorbar for the ContourSet returned by the contourf call.
cbar = plt.colorbar(CS)
cbar.ax.set_ylabel('verbosity coefficient')
# Add the contour line levels to the colorbar
cbar.add_lines(CS2)

plt.figure()

# Now make a contour plot with the levels specified,
# and with the colormap generated automatically from a list
# of colors.
levels = [-1.5, -1, -0.5, 0, 0.5, 1]
CS3 = plt.contourf(X, Y, Z, levels,
                  colors = ('r', 'g', 'b'),
                  origin=origin,
                  extend='both')

# Our data range extends outside the range of levels; make
# data below the lowest contour level yellow, and above the
# highest level cyan:
CS3.cmap.set_under('yellow')
CS3.cmap.set_over('cyan')

CS4 = plt.contour(X, Y, Z, levels,
                  colors = ('k',),
                  linewidths = (3,),
origin = origin
plt.title('Listed colors (3 masked regions)')
plt.clabel(CS4, fmt = '%2.1f', colors = 'w', fontsize=14)

# Notice that the colorbar command gets all the information it
# needs from the ContourSet object, CS3.
plt.colorbar(CS3)

# Illustrate all 4 possible "extend" settings:
extends = ['neither', 'both', 'min', 'max']
cmap = plt.cm.get_cmap("winter")
cmap.set_under("magenta")
cmap.set_over("yellow")
# Note: contouring simply excludes masked or nan regions, so
# instead of using the "bad" colormap value for them, it draws
# nothing at all in them. Therefore the following would have
# no effect:
#cmap.set_under("red")

fig, axs = plt.subplots(2,2)
for ax, extend in zip(axs.ravel(), extends):
    cs = ax.contourf(X, Y, Z, levels, cmap=cmap, extend=extend, origin=origin)
    fig.colorbar(cs, ax=ax, shrink=0.9)
    ax.set_title("extend = %s" % extend)
    ax.locator_params(nbins=4)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.41 `pylab_examples` example code: `contourf_hatching.py`
```python
import matplotlib.pyplot as plt
import numpy as np

# invent some numbers, turning the x and y arrays into simple
# 2d arrays, which make combining them together easier.
x = np.linspace(-3, 5, 150).reshape(1, -1)
y = np.linspace(-3, 5, 120).reshape(-1, 1)
z = np.cos(x) + np.sin(y)

# we no longer need x and y to be 2 dimensional, so flatten them.
x, y = x.flatten(), y.flatten()

# ---------------------------------------------
# | Plot #1 |
# ---------------------------------------------
# the simplest hatched plot with a colorbar
fig = plt.figure()
cs = plt.contourf(x, y, z, hatches=['-', '/', '\', '//'],
                  cmap=plt.get_cmap('gray'),
                  extend='both', alpha=0.5
              )
plt.colorbar()
```

84.41. pylab_examples example code: contourf_hatching.py 2015
# a plot of hatches without color with a legend
plt.figure()
n_levels = 6
plt.contour(x, y, z, n_levels, colors='black', linestyles='--')
cs = plt.contourf(x, y, z, n_levels, colors='none',
    hatches=['.', '/', '\', None, '\\', '*'],
    extend='lower'
 )

# create a legend for the contour set
artists, labels = cs.legend_elements()
plt.legend(artists, labels, handleheight=2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demonstrate use of a log color scale in contourf

```python
from matplotlib import pyplot as P
import numpy as np
from numpy import ma
from matplotlib import colors, ticker, cm
from matplotlib.mlab import bivariate_normal

N = 100
x = np.linspace(-3.0, 3.0, N)
y = np.linspace(-2.0, 2.0, N)

X, Y = np.meshgrid(x, y)

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.

z = (bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0)
     + 0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0))
```

---

84.42. `pylab_examples` example code: `contourf_log.py`
# Put in some negative values (lower left corner) to cause trouble with logs:
z[:5, :5] = -1

# The following is not strictly essential, but it will eliminate
# a warning. Comment it out to see the warning.
z = ma.masked_where(z<= 0, z)

# Automatic selection of levels works; setting the
# log locator tells contourf to use a log scale:
cs = P.contourf(X, Y, z, locator=ticker.LogLocator(), cmap=cm.PuBu_r)

# Alternatively, you can manually set the levels
# and the norm:
#lev_exp = np.arange(np.floor(np.log10(z.min()))-1),
# np.ceil(np.log10(z.max()))+1))
#levs = np.power(10, lev_exp)
#cs = P.contourf(X, Y, z, levs, norm=colors.LogNorm())

# The 'extend' kwarg does not work yet with a log scale.

cbar = P.colorbar()

P.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
An example of how to interact with the plotting canvas by connecting to move and click events
"""

from __future__ import print_function
import sys
import *

fig, ax = plt.subplots()
t = arange(0.0, 1.0, 0.01)
s = sin(2*pi*t)
ax.plot(t,s)

def on_move(event):
    # get the x and y pixel coords
    x, y = event.x, event.y

    if event.inaxes:
        ax = event.inaxes  # the axes instance
```python
print ('data coords %f %f' % (event.xdata, event.ydata))

def on_click(event):
    # get the x and y coords, flip y from top to bottom
    x, y = event.x, event.y
    if event.button==1:
        if event.inaxes is not None:
            print ('data coords %f %f' % (event.xdata, event.ydata))

binding_id = connect('motion_notify_event', on_move)
connect('button_press_event', on_click)

if "test_disconnect" in sys.argv:
    print ('disconnecting console coordinate printout...')
    disconnect(binding_id)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.44 pylab_examples example code: coords_report.py
#!/usr/bin/env python

# override the default reporting of coords

from pylab import *

def millions(x):
    return '$%1.1fM' % (x*1e-6)

x = rand(20)
y = 1e7*rand(20)

fig, ax = subplots()
ax.fmt_ydata = millions
plot(x, y, 'o')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
Compute the cross spectral density of two signals
"""

import numpy as np
import matplotlib.pyplot as plt

# make a little extra space between the subplots
plt.subplots_adjust(wspace=0.5)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))  # white noise 1
nse2 = np.random.randn(len(t))  # white noise 2
r = np.exp(-t/0.05)

cnse1 = np.convolve(nse1, r, mode='same')*dt  # colored noise 1
cnse2 = np.convolve(nse2, r, mode='same')*dt  # colored noise 2

# two signals with a coherent part and a random part
s1 = 0.01*np.sin(2*np.pi*10*t) + cnse1
s2 = 0.01*np.sin(2*np.pi*10*t) + cnse2

plt.subplot(211)
plt.plot(t, s1, 'b-', t, s2, 'g-')
plt.xlim(0, 5)
plt.xlabel('time')
plt.ylabel('s1 and s2')
plt.grid(True)

plt.subplot(212)
cxy, f = plt.csd(s1, s2, 256, 1./dt)
plt.ylabel('CSD (db)')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.46 pylab_examples example code: cursor_demo.py

[source code]

#!/usr/bin/env python
# -*- noplot -*-
#-=-
#-=-
#-=-

"""
This example shows how to use matplotlib to provide a data cursor. It uses matplotlib
to draw the cursor and may be a slow since this requires redrawing the figure with
every mouse move.
"""
Faster cursoring is possible using native GUI drawing, as in
wxcursor_demo.py

```python
from __future__ import print_function
from pylab import *

class Cursor:
    def __init__(self, ax):
        self.ax = ax
        self.lx = ax.axhline(color='k')  # the horiz line
        self.ly = ax.axvline(color='k')  # the vert line

        # text location in axes coords
        self.txt = ax.text(0.7, 0.9, '', transform=ax.transAxes)

    def mouse_move(self, event):
        if not event.inaxes: return

        x, y = event.xdata, event.ydata
        # update the line positions
        self.lx.set_ydata(y)
        self.ly.set_xdata(x)

        self.txt.set_text('x=%1.2f, y=%1.2f' % (x, y))
        draw()


class SnaptoCursor:
    
    Like Cursor but the crosshair snaps to the nearest x,y point
    For simplicity, I'm assuming x is sorted

    def __init__(self, ax, x, y):
        self.ax = ax
        self.lx = ax.axhline(color='k')  # the horiz line
        self.ly = ax.axvline(color='k')  # the vert line
        self.x = x
        self.y = y

        # text location in axes coords
        self.txt = ax.text(0.7, 0.9, '', transform=ax.transAxes)

    def mouse_move(self, event):
        if not event.inaxes: return

        x, y = event.xdata, event.ydata

        indx = searchsorted(self.x, [x])[0]
        x = self.x[indx]
        y = self.y[indx]

        # update the line positions
        self.lx.set_ydata(y)
```

```
self.ly.set_xdata(x)

self.txt.set_text( 'x=%1.2f, y=%1.2f'%(x,y) )
print ('x=%1.2f, y=%1.2f'%(x,y))
draw()

t = arange(0.0, 1.0, 0.01)
s = sin(2*pi*t)
fig, ax = subplots()

cursor = Cursor(ax)
#cursor = SnaptoCursor(ax, t, s)
connect('motion_notify_event', cursor.mouse_move)

ax.plot(t, s, 'o')
axis([0,1,-1,1])
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Custom Blue-Red colormaps

Alpha = 1

Varying alpha
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

""
Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

cdict = {'red': ((0.0, 0.0, 0.0),
                 (0.5, 1.0, 1.0),
                 (1.0, 1.0, 1.0)),

           'green': ((0.0, 0.0, 0.0),
                      (0.25, 0.0, 0.0),
                      (0.75, 1.0, 1.0),
                      (1.0, 1.0, 1.0)),

           'blue': ((0.0, 0.0, 0.0),
                     (0.5, 0.0, 0.0),
                     (1.0, 1.0, 1.0))}

If, as in this example, there are no discontinuities in the r, g, and b components, then it is quite simple: the second and third element of each tuple, above, is the same—call it "y". The first element ("x") defines interpolation intervals over the full range of 0 to 1, and it must span that whole range. In other words, the values of x divide the 0-to-1 range into a set of segments, and y gives the end-point color values for each segment.

Now consider the green. cdict['green'] is saying that for 0 <= x <= 0.25, y is zero; no green.
0.25 < x <= 0.75, y varies linearly from 0 to 1.
x > 0.75, y remains at 1, full green.

If there are discontinuities, then it is a little more complicated. Label the 3 elements in each row in the cdict entry for a given color as (x, y0, y1). Then for values of x between x[i] and x[i+1] the color value is interpolated between y1[i] and y0[i+1].

Going back to the cookbook example, look at cdict['red']; because y0 != y1, it is saying that for x from 0 to 0.5, red increases from 0 to 1, but then it jumps down, so that for x from 0.5 to 1, red increases from 0.7 to 1. Green ramps from 0 to 1 as x goes from 0 to 0.5, then jumps back to 0, and ramps back to 1 as x goes from 0.5 to 1.

row i:  x  y0  y1
       /   /   /
row i+1: x  y0  y1
Above is an attempt to show that for x in the range $x[i]$ to $x[i+1]$, the interpolation is between $y1[i]$ and $y0[i+1]$. So, $y0[0]$ and $y1[-1]$ are never used.

```python
cdict1 = {'red': ((0.0, 0.0, 0.0),
                (0.5, 0.0, 0.1),
                (1.0, 1.0, 1.0)),
          'green': ((0.0, 0.0, 0.0),
                (1.0, 0.0, 0.0)),
          'blue': ((0.0, 0.0, 1.0),
                (0.5, 0.1, 0.0),
                (1.0, 0.0, 0.0))}

cdict2 = {'red': ((0.0, 0.0, 0.0),
                (0.5, 0.0, 1.0),
                (1.0, 0.1, 1.0)),
          'green': ((0.0, 0.0, 0.0),
                (1.0, 0.0, 0.0)),
          'blue': ((0.0, 0.0, 1.0),
                (0.5, 1.0, 0.0),
                (1.0, 0.0, 0.0))}

cdict3 = {'red': ((0.0, 0.0, 0.0),
                (0.25, 0.0, 0.0),
                (0.5, 0.8, 1.0),
                (0.75, 1.0, 1.0),
                (1.0, 0.4, 1.0)),
          'green': ((0.0, 0.0, 0.0),
                (0.25, 0.0, 0.0),
                (0.5, 0.9, 0.9),
                (0.75, 0.0, 0.0),
                (1.0, 0.0, 0.0)),
          'blue': ((0.0, 0.0, 0.4),
                (0.25, 1.0, 1.0),
                (0.5, 1.0, 0.8),
                (0.75, 0.0, 0.0),
                (1.0, 0.0, 0.0))}

# Make a modified version of cdict3 with some transparency
# in the middle of the range.
```
Matplotlib, Release 1.4.3

```python
cdict4 = cdict3.copy()
cdict4['alpha'] = ((0.0, 1.0, 1.0),
                   # (0.25,1.0, 1.0),
                   (0.5, 0.3, 0.3),
                   # (0.75,1.0, 1.0),
                   (1.0, 1.0, 1.0))

# Now we will use this example to illustrate 3 ways of
# handling custom colormaps.
# First, the most direct and explicit:
blue_red1 = LinearSegmentedColormap('BlueRed1', cdict)

# Second, create the map explicitly and register it.
# Like the first method, this method works with any kind
# of Colormap, not just
# a LinearSegmentedColormap:
blue_red2 = LinearSegmentedColormap('BlueRed2', cdict)
plt.register_cmap(cmap=blue_red2)

# Third, for LinearSegmentedColormap only,
# leave everything to register_cmap:
plt.register_cmap(name='BlueRed3', data=cdict3)  # optional lut kwarg
plt.register_cmap(name='BlueRedAlpha', data=cdict4)

# Make some illustrative fake data:
x = np.arange(0, np.pi, 0.1)
y = np.arange(0, 2*np.pi, 0.1)
X, Y = np.meshgrid(x,y)
Z = np.cos(X) * np.sin(Y) * 10

# Make the figure:
plt.figure(figsize=(6,9))
plt.subplots_adjust(left=0.02, bottom=0.06, right=0.95, top=0.94, wspace=0.05)

# Make 4 subplots:
plt.subplot(2,2,1)
plt.imshow(Z, interpolation='nearest', cmap=blue_red1)
plt.colorbar()

plt.subplot(2,2,2)
cmap = plt.get_cmap('BlueRed2')
plt.imshow(Z, interpolation='nearest', cmap=cmap)
plt.colorbar()

# Now we will set the third cmap as the default. One would
# not normally do this in the middle of a script like this;
```

84.47. pylab_examples example code: custom_cmap.py 2029
# it is done here just to illustrate the method.

```python
plt.rcParams['image.cmap'] = 'BlueRed3'

plt.subplot(2,2,3)
plt.imshow(Z, interpolation='nearest')
plt.colorbar()
plt.title("Alpha = 1")
```

# Or as yet another variation, we can replace the rcParams specification *before* the imshow with the following *after* imshow.
# This sets the new default *and* sets the colormap of the last image-like item plotted via pyplot, if any.

```python
plt.subplot(2,2,4)
# Draw a line with low zorder so it will be behind the image.
plt.plot([0, 10*np.pi], [0, 20*np.pi], color='c', lw=20, zorder=-1)
plt.imshow(Z, interpolation='nearest')
plt.colorbar()
```

# Here it is: changing the colormap for the current image and its colorbar after they have been plotted.
```python
plt.set_cmap('BlueRedAlpha')
plt.title("Varying alpha")
```

```python
plt.suptitle('Custom Blue-Red colormaps', fontsize=16)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
You can pass a custom Figure constructor to figure if you want to derive from the default Figure. This simple example creates a figure with a figure title

```python
from matplotlib.pyplot import figure, show
from matplotlib.figure import Figure

class MyFigure(Figure):
    def __init__(self, *args, **kwargs):
        """
        custom kwarg figtitle is a figure title
        """
        figtitle = kwargs.pop('figtitle', 'hi mom')
        Figure.__init__(self, *args, **kwargs)
        self.text(0.5, 0.95, figtitle, ha='center')

fig = figure(FigureClass=MyFigure, figtitle='my title')
ap = fig.add_subplot(111)
ap.plot([1,2,3])

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
The new ticker code was designed to explicitly support user customized ticking. The documentation http://matplotlib.org/matplotlib.ticker.html details this process. That code defines a lot of preset tickers but was primarily designed to be user extensible.

In this example a user defined function is used to format the ticks in millions of dollars on the y axis

```python
#!/usr/bin/env python

""

The two args are the value and tick position
```

```python
def millions(x, pos):
    'The two args are the value and tick position'
    return '$%1.1fM' % (x*1e-6)
```

```python
from matplotlib.ticker import FuncFormatter
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(4)
money = [1.5e5, 2.5e6, 5.5e6, 2.0e7]

```
formatter = FuncFormatter(millions)

fig, ax = plt.subplots()
ax.yaxis.set_major_formatter(formatter)
plt.bar(x, money)
plt.xticks(x + 0.5, ('Bill', 'Fred', 'Mary', 'Sue'))
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.50  pylab_examples example code: customize_rc.py

""
I'm not trying to make a good looking figure here, but just to show some examples of customizing rc params on the fly

If you like to work interactively, and need to create different sets of defaults for figures (e.g., one set of defaults for publication, one set for interactive exploration), you may want to define some functions in a custom module that set the defaults, e.g.,

def set_pub():

84.50.  pylab_examples example code: customize_rc.py  2033
rc('font', weight='bold')  # bold fonts are easier to see
rc('tick', labelsize=15)  # tick labels bigger
rc('lines', lw=1, color='k')  # thicker black lines (no budget for color!)
rc('grid', c='0.5', ls='-', lw=0.5)  # solid gray grid lines
rc('savefig', dpi=300)  # higher res outputs

Then as you are working interactively, you just need to do

```python
>>> set_pub()
>>> subplot(111)
>>> plot([1,2,3])
>>> savefig('myfig')
>>> rcdefaults()  # restore the defaults
```

```python
from pylab import *
```

```
subplot(311)
plot([1,2,3])

# the axes attributes need to be set before the call to subplot
rc('font', weight='bold')
rc('xtick.major', size=5, pad=7)
rc('xtick', labelsize=15)

# using aliases for color, linestyle and linewidth; gray, solid, thick
rc('grid', c='0.5', ls='-', lw=5)
rc('lines', lw=2, color='g')
subplot(312)

plot([1,2,3])
grid(True)

rcdefaults()
subplot(313)
plot([1,2,3])
grid(True)
show()
```

Keywords: python, matplotlib, pylab, example, codex (see `Search examples`)
import matplotlib.pyplot as plt

DATA = [(1, 3),
        (2, 4),
        (3, 1),
        (4, 2)]

# dash_style =
# (direction, length, (text)rotation, dashrotation, push
# (The parameters are varied to show their effects,
# not for visual appeal).

fig, ax = plt.subplots()

for data in zip(*DATA):
    ax.plot(*data, marker='o')

(1, 3)
(2, 4)
(3, 1)
(4, 2)
(x, y) = DATA[i]
(dd, dl, r, dr, dp) = dash_style[i]
# print 'dashlen call', dl

t = ax.text(x, y, str((x, y)), withdash=True,
    dashdirection=dd,
    dashlength=dl,
    rotation=r,
    dashrotation=dr,
    dashpush=dp,
    )

ax.set_xlim((0.0, 5.0))
ax.set_ylim((0.0, 5.0))

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.52 pylab_examples example code: data_helper.py

#!/usr/bin/env python
# Some functions to load a return data for the plot demos

from numpy import fromstring, argsort, take, array, resize
import matplotlib.cbook as cbook

def get_two_stock_data():
    """
    load stock time and price data for two stocks The return values
    (d1, p1, d2, p2) are the trade time (in days) and prices for stocks 1
    and 2 (intc and aapl)
    """
    ticker1, ticker2 = 'INTC', 'AAPL'

    file1 = cbook.get_sample_data('INTC.dat.gz')
    file2 = cbook.get_sample_data('AAPL.dat.gz')
    M1 = fromstring( file1.read(), '<d' )
    M1 = resize(M1, (M1.shape[0]/2,2) )

    M2 = fromstring( file2.read(), '<d' )
    M2 = resize(M2, (M2.shape[0]/2,2) )

    d1, p1 = M1[:,0], M1[:,1]
    d2, p2 = M2[:,0], M2[:,1]
    return (d1, p1, d2, p2)

def get_daily_data():
    """
    return stock1 and stock2 instances, each of which have attributes
    """
open, high, low, close, volume

as numeric arrays

class C: pass

def get_ticker(ticker):
    vals = []

    datafile = cbook.get_sample_data('%s.csv' % ticker, asfileobj=False)

    lines = open(datafile).readlines()
    for line in lines[1:]:
        vals.append([float(val) for val in line.split(',')[1:]]

    M = array(vals)
    c = C()
    c.open = M[:, 0]
    c.high = M[:, 1]
    c.low = M[:, 2]
    c.close = M[:, 3]
    c.volume = M[:, 4]
    return c

c1 = get_ticker('intc')
c2 = get_ticker('msft')
return c1, c2

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Show how to make date plots in matplotlib using date tick locators and formatters. See major_minor_demo1.py for more information on controlling major and minor ticks.

All matplotlib date plotting is done by converting date instances into days since the 0001-01-01 UTC. The conversion, tick locating and formatting is done behind the scenes so this is most transparent to you. The dates module provides several converter functions date2num and num2date.

This example requires an active internet connection since it uses yahoo finance to get the data for plotting.

```python
#!/usr/bin/env python

""
Show how to make date plots in matplotlib using date tick locators and formatters. See major_minor_demo1.py for more information on controlling major and minor ticks.

All matplotlib date plotting is done by converting date instances into days since the 0001-01-01 UTC. The conversion, tick locating and formatting is done behind the scenes so this is most transparent to you. The dates module provides several converter functions date2num and num2date.

This example requires an active internet connection since it uses yahoo finance to get the data for plotting.

import matplotlib.pyplot as plt
from matplotlib.finance import quotes_historical_yahoo_ochl
from matplotlib.dates import YearLocator, MonthLocator, DateFormatter
import datetime

date1 = datetime.date(1995, 1, 1)
date2 = datetime.date(2004, 4, 12)
```

```python
import matplotlib.pyplot as plt
from matplotlib.finance import quotes_historical_yahoo_ochl
from matplotlib.dates import YearLocator, MonthLocator, DateFormatter
import datetime

date1 = datetime.date(1995, 1, 1)
date2 = datetime.date(2004, 4, 12)
```
years = YearLocator()  # every year
months = MonthLocator()  # every month
yearsFmt = DateFormatter('%Y')

quotes = quotes_historical_yahoo_ochl('INTC', date1, date2)
if len(quotes) == 0:
    raise SystemExit

dates = [q[0] for q in quotes]
opens = [q[1] for q in quotes]

fig, ax = plt.subplots()
ax.plot_date(dates, opens, '-')

# format the ticks
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(yearsFmt)
ax.xaxis.set_minor_locator(months)
ax.autoscale_view()

# format the coords message box
def price(x):
    return '$%1.2f' % x
ax.fmt_xdata = DateFormatter('%Y-%m-%d')
ax.fmt_ydata = price
ax.grid(True)

fig.autofmt_xdate()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Show how to make date plots in matplotlib using date tick locators and formatters. See major_minor_demo1.py for more information on controlling major and minor ticks.

```python
#!/usr/bin/env python

""
Show how to make date plots in matplotlib using date tick locators and formatters. See major_minor_demo1.py for more information on controlling major and minor ticks
""

from __future__ import print_function
import datetime
import matplotlib.pyplot as plt
from matplotlib.dates import MONDAY
from matplotlib.finance import quotes_historical_yahoo_ochl
from matplotlib.dates import MonthLocator, WeekdayLocator, DateFormatter

date1 = datetime.date(2002, 1, 5)
date2 = datetime.date(2003, 12, 1)

date1 = datetime.date(2002, 1, 5)
date2 = datetime.date(2003, 12, 1)

# every monday
mondays = WeekdayLocator(MONDAY)

# every 3rd month
```
months = MonthLocator(range(1, 13), bymonthday=1, interval=3)
monthsFmt = DateFormatter("%b %Y")

quotes = quotes_historical_yahoo_ochl('INTC', date1, date2)
if len(quotes) == 0:
    print('Found no quotes')
    raise SystemExit

dates = [q[0] for q in quotes]
opens = [q[1] for q in quotes]

fig, ax = plt.subplots()
ax.plot_date(dates, opens, '-')
ax.xaxis.set_major_locator(months)
ax.xaxis.set_major_formatter(monthsFmt)
ax.xaxis.set_minor_locator(mondays)
ax.autoscale_view()
#ax.xaxis.grid(False, 'major')
#ax.xaxis.grid(True, 'minor')
ax.grid(True)

fig.autofmt_xdate()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
#!/usr/bin/env python

import datetime
import matplotlib.pyplot as plt
from matplotlib.dates import DayLocator, HourLocator, DateFormatter, drange
from numpy import arange

date1 = datetime.datetime(2000, 3, 2)
date2 = datetime.datetime(2000, 3, 6)
delta = datetime.timedelta(hours=6)
dates = drange(date1, date2, delta)
y = arange(len(dates)*1.0)

fig, ax = plt.subplots()
ax.plot_date(dates, y*y)

# this is superfluous, since the autoscaler should get it right, but
# use date2num and num2date to convert between dates and floats if
# you want; both date2num and num2date convert an instance or sequence
ax.set_xlim(dates[0], dates[-1])
```
# The hour locator takes the hour or sequence of hours you want to
# tick, not the base multiple

tax.xaxis.set_major_locator(DayLocator())
tax.xaxis.set_minor_locator(HourLocator(arange(0,25,6)))
tax.xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'))

tax.fmt_xdata = DateFormatter('%Y-%m-%d %H:%M:%S')
fig.autofmt_xdate()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.56  pylab_examples example code: date_demo_rrule.py

#!/usr/bin/env python

'''
Show how to use an rrule instance to make a custom date ticker - here
we put a tick mark on every 5th easter

See https://moin.conectiva.com.br/DateUtil for help with rrules

84.56.  pylab_examples example code: date_demo_rrule.py  2043
"""
import matplotlib.pyplot as plt
from matplotlib.dates import YEARLY, DateFormatter, rrulewrapper, RRuleLocator, drange
import numpy as np
import datetime

# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
formatter = DateFormatter('%m/%d/%y')
date1 = datetime.date(1952, 1, 1)
date2 = datetime.date(2004, 4, 12)
delta = datetime.timedelta(days=100)

dates = drange(date1, date2, delta)
s = np.random.rand(len(dates))  # make up some random y values

fig, ax = plt.subplots()
plt.plot_date(dates, s)
ax.xaxis.set_major_locator(loc)
ax.xaxis.set_major_formatter(formatter)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=30, fontsize=10)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)"""
When plotting daily data, a frequent request is to plot the data ignoring skips, e.g., no extra spaces for weekends. This is particularly common in financial time series, when you may have data for M-F and not Sat, Sun and you don’t want gaps in the x axis. The approach is to simply use the integer index for the xdata and a custom tick Formatter to get the appropriate date string for a given index.

```
from __future__ import print_function
import numpy
from matplotlib.mlab import csv2rec
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
from matplotlib.ticker import Formatter

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading %s ' % datafile)
r = csv2rec(datafile)[-40:]

class MyFormatter(Formatter):
    def __init__(self, dates, fmt='%Y-%m-%d'):
```


```
self.dates = dates
self.fmt = fmt

def __call__(self, x, pos=0):
    'Return the label for time x at position pos'
    ind = int(round(x))
    if ind>=len(self.dates) or ind<0: return ''

    return self.dates[ind].strftime(self.fmt)

formatter = MyFormatter(r.date)

fig, ax = plt.subplots()
ax.xaxis.set_major_formatter(formatter)
ax.plot(numpy.arange(len(r)), r.close, 'o-')
fig.autofmt_xdate()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab

def smooth1d(x, window_len):
    # copied from http://www.scipy.org/Cookbook/SignalSmooth

    s=np.r_[2*x[0]-x[window_len:1:-1],x,2*x[-1]-x[-1:-window_len:-1]]
    w = np.hanning(window_len)
    y=np.convolve(w/w.sum(),s,mode='same')
    return y[window_len-1:-window_len+1]
def smooth2d(A, sigma=3):
    window_len = max(int(sigma), 3)*2+1
    A1 = np.array([smooth1d(x, window_len) for x in np.asarray(A)])
    A2 = np.transpose(A1)
    A3 = np.array([smooth1d(x, window_len) for x in A2])
    A4 = np.transpose(A3)
    return A4

class BaseFilter(object):
    def prepare_image(self, src_image, dpi, pad):
        ny, nx, depth = src_image.shape
        # tgt_image = np.zeros([pad*2+ny, pad*2+nx, depth], dtype="d")
        padded_src = np.zeros([pad*2+ny, pad*2+nx, depth], dtype="d")
        padded_src[pad:-pad, pad:-pad, :] = src_image[:, :, :]
        return padded_src#

def get_pad(self, dpi):
    return 0

def __call__(self, im, dpi):
    pad = self.get_pad(dpi)
    padded_src = self.prepare_image(im, dpi, pad)
    tgt_image = self.process_image(padded_src, dpi)
    return tgt_image, -pad, -pad

class OffsetFilter(BaseFilter):
    def __init__(self, offsets=None):
        if offsets is None:
            self.offsets = (0, 0)
        else:
            self.offsets = offsets

def get_pad(self, dpi):
    return int(max(*self.offsets)/72.*dpi)

def process_image(self, padded_src, dpi):
    ox, oy = self.offsets
    a1 = np.roll(padded_src, int(ox/72.*dpi), axis=1)
    a2 = np.roll(a1, -int(oy/72.*dpi), axis=0)
    return a2

class GaussianFilter(BaseFilter):
    "simple gauss filter"
    def __init__(self, sigma, alpha=0.5, color=None):
        self.sigma = sigma
        self.alpha = alpha
if color is None:
    self.color=(0, 0, 0)
else:
    self.color=color

def get_pad(self, dpi):
    return int(self.sigma*3/72.*dpi)

def process_image(self, padded_src, dpi):
    #offsetx, offsety = int(self.offsets[0]), int(self.offsets[1])
    tgt_image = np.zeros_like(padded_src)
    aa = smooth2d(padded_src[:,:,-1]*self.alpha,
                  self.sigma/72.*dpi)
    tgt_image[:,:,-1] = aa
    tgt_image[:,:,:-1] = self.color
    return tgt_image

class DropShadowFilter(BaseFilter):
    def __init__(self, sigma, alpha=0.3, color=None, offsets=None):
        self.gauss_filter = GaussianFilter(sigma, alpha, color)
        self.offset_filter = OffsetFilter(offsets)

    def get_pad(self, dpi):
        return max(self.gauss_filter.get_pad(dpi),
                   self.offset_filter.get_pad(dpi))

    def process_image(self, padded_src, dpi):
        t1 = self.gauss_filter.process_image(padded_src, dpi)
        t2 = self.offset_filter.process_image(t1, dpi)
        return t2

class LightFilter(BaseFilter):
    "simple gauss filter"
    def __init__(self, sigma, fraction=0.5):
        self.gauss_filter = GaussianFilter(sigma, alpha=1)
        self.light_source = LightSource()
        self.fraction = fraction
        #hsv_min_val=0.5,hsv_max_val=0.9,
        #hsv_min_sat=0.1,hsv_max_sat=0.1)

    def get_pad(self, dpi):
        return self.gauss_filter.get_pad(dpi)

    def process_image(self, padded_src, dpi):
        t1 = self.gauss_filter.process_image(padded_src, dpi)
        elevation = t1[:,:,3]
        rgb = padded_src[:,:,:3]
        rgb2 = self.light_source.shade_rgb(rgb, elevation,
                                            fraction=self.fraction)
tgt = np.empty_like(padded_src)
tgt[:,:,3] = rgb2
tgt[:,:,3] = padded_src[:,:,3]

return tgt

class GrowFilter(BaseFilter):
    "enlarge the area"
def __init__(self, pixels, color=None):
    self.pixels = pixels
    if color is None:
        self.color=(1, 1, 1)
    else:
        self.color=color
def __call__(self, im, dpi):
    pad = self.pixels
    ny, nx, depth = im.shape
    new_im = np.empty([pad*2+ny, pad*2+nx, depth], dtype="d")
    alpha = new_im[:,:,3]
    alpha.fill(0)
    alpha[pad:-pad, pad:-pad] = im[:,:,-1]
    alpha2 = np.clip(smooth2d(alpha, self.pixels/72.*dpi) * 5, 0, 1)
    new_im[:,:,3] = alpha2
    new_im[:,:,-1] = self.color
    offsetx, offsety = -pad, -pad

    return new_im, offsetx, offsety

from matplotlib.artist import Artist
class FilteredArtistList(Artist):
    """
    A simple container to draw filtered artist.
    """
    def __init__(self, artist_list, filter):
        self._artist_list = artist_list
        self._filter = filter
        Artist.__init__(self)
    def draw(self, renderer):
        renderer.start_rasterizing()
        renderer.start_filter()
        for a in self._artist_list:
            a.draw(renderer)
        renderer.stop_filter(self._filter)
        renderer.stop_rasterizing()
import matplotlib.transforms as mtransforms

def filtered_text(ax):
    # mostly copied from contour_demo.py

    # prepare image
    delta = 0.025
    x = np.arange(-3.0, 3.0, delta)
    y = np.arange(-2.0, 2.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
    Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
    # difference of Gaussians
    Z = 10.0 * (Z2 - Z1)

    # draw
    im = ax.imshow(Z, interpolation='bilinear', origin='lower',
                   cmap=cm.gray, extent=(-3,3,-2,2))
    levels = np.arange(-1.2, 1.6, 0.2)
    CS = ax.contour(Z, levels,
                    origin='lower',
                    linewidths=2,
                    extent=(-3,3,-2,2))
    ax.set_aspect("auto")

    # contour label
    cl = ax.clabel(CS, levels[1::2],
                   inline=1,
                   fmt="%.1f",
                   fontsize=11)
    # change clable color to black
    from matplotlib.pathEffects import Normal
    for t in cl:
        t.set_color("k")
        t.set_path_effects([Normal()])
    # Add white glows to improve visibility of labels.
    white_glow = FilteredArtistList(cl, GrowFilter(3))
    ax.add_artist(white_glow)
    white_glow.set_zorder(cl[0].get_zorder()-0.1)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

    def drop_shadow_line(ax):
        # copied from examples/misc/svg_filter_line.py

        # draw lines
        l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-")
gauss = DropShadowFilter(4)

for l in [l1, l2]:
    # draw shadows with same lines with slight offset.
    xx = l.get_xdata()
    yy = l.get_ydata()
    shadow, = ax.plot(xx, yy)
    shadow.update_from(l)

    # offset transform
    ot = mtransforms.offset_copy(l.get_transform(), ax.figure,
                               x=4.0, y=-6.0, units='points')
    shadow.set_transform(ot)
    shadow.set_zorder(l.get_zorder()-0.5)
    shadow.set_agg_filter(gauss)
    shadow.set_rasterized(True)

ax.set_xlim(0., 1.)
ax.set_ylim(0., 1.)

ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)


def drop_shadow_patches(ax):
    # copied from barchart_demo.py
    N = 5
    menMeans = (20, 35, 30, 35, 27)

    ind = np.arange(N)  # the x locations for the groups
    width = 0.35  # the width of the bars

    rects1 = ax.bar(ind, menMeans, width, color='r', ec="w", lw=2)

    womenMeans = (25, 32, 34, 20, 25)
    rects2 = ax.bar(ind+width+0.1, womenMeans, width, color='y', ec="w", lw=2)
# gauss = GaussianFilter(1.5, offsets=(1,1), )
gauss = DropShadowFilter(5, offsets=(1,1), )
shadow = FilteredArtistList(rects1+rects2, gauss)
ax.add_artist(shadow)
shadow.set_zorder(rects1[0].get_zorder()-0.1)

ax.set_xlim(ind[0]-0.5, ind[-1]+1.5)
ax.set_ylim(0, 40)

ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

def light_filter_pie(ax):
    fracs = [15,30,45, 10]
    explode=(0, 0.05, 0, 0)
    pies = ax.pie(fracs, explode=explode)
    ax.patch.set_visible(True)

    light_filter = LightFilter(9)
    for p in pies[0]:
        p.set_agg_filter(light_filter)
        p.set_rasterized(True)  # to support mixed-mode renderers
        p.set(ec="none",
             lw=2)

    gauss = DropShadowFilter(9, offsets=(3,4), alpha=0.7)
    shadow = FilteredArtistList(pies[0], gauss)
    ax.add_artist(shadow)
    shadow.set_zorder(pies[0][0].get_zorder()-0.1)

if 1:
    plt.figure(1, figsize=(6, 6))
    plt.subplots_adjust(left=0.05, right=0.95)

    ax = plt.subplot(221)
    filtered_text(ax)

    ax = plt.subplot(222)
    drop_shadow_line(ax)

    ax = plt.subplot(223)
    drop_shadow_patches(ax)

    ax = plt.subplot(224)
    ax.set_aspect(1)
    light_filter_pie(ax)
    ax.set_frame_on(True)

    plt.show()
84.59  pylab_examples example code: demo_annotation_box.py

```python
import matplotlib.pyplot as plt
from matplotlib.offsetbox import TextArea, DrawingArea, OffsetImage, AnnotationBbox
from matplotlib.cbook import get_sample_data

import numpy as np

if 1:
    fig, ax = plt.subplots()

    offsetbox = TextArea("Test 1", minimumdescent=False)

    xy = (0.5, 0.7)

    ax.plot(xy[0], xy[1], ".r")

    ab = AnnotationBbox(offsetbox, xy,
                        xybox=(-20, 40),
                        xycoords='data',)
```

boxcoords="offset points",
arrowprops=dict(arrowstyle="->"))

ax.add_artist(ab)

offsetbox = TextArea("Test", minimumdescent=False)

ab = AnnotationBbox(offsetbox, xy,
                           xybox=(1.02, xy[1]),
                           xycoords='data',
                           boxcoords=('axes fraction', "data"),
                           box_alignment=(0.,0.5),
                           arrowprops=dict(arrowstyle="->"))

ax.add_artist(ab)

from matplotlib.patches import Circle
da = DrawingArea(20, 20, 0, 0)
p = Circle((10, 10), 10)
da.add_artist(p)

xy = [0.3, 0.55]
ab = AnnotationBbox(da, xy,
                           xybox=(1.02, xy[1]),
                           xycoords='data',
                           boxcoords=('axes fraction', "data"),
                           box_alignment=(0.,0.5),
                           arrowprops=dict(arrowstyle="->")))

ax.add_artist(ab)

arr = np.arange(100).reshape((10,10))

im = OffsetImage(arr, zoom=2)

ab = AnnotationBbox(im, xy,
                           xybox=(-50., 50.),
                           xycoords='data',
                           boxcoords="offset points",
                           pad=0.3,
                           arrowprops=dict(arrowstyle="->")))

ax.add_artist(ab)

# another image

from matplotlib._png import read_png

fn = get_sample_data("grace_hopper.png", asfileobj=False)

arr_lena = read_png(fn)
imagebox = OffsetImage(arr_lena, zoom=0.2)

ab = AnnotationBbox(imagebox, xy,
    xybox=(120., -80.),
    xycoords='data',
    boxcoords="offset points",
    pad=0.5,
    arrowprops=*
    connectionstyle="angle,angleA=0,angleB=90,rad=3"
    
)

ax.add_artist(ab)

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.image import BboxImage
from matplotlib.transforms import Bbox, TransformedBbox

if __name__ == '__main__':
    fig = plt.figure(1)
    ax = plt.subplot(121)

    txt = ax.text(0.5, 0.5, "test", size=30, ha="center", color="w")
    kwargs = dict()
    bbox_image = BboxImage(txt.get_window_extent,
                           norm = None,
                           origin=None,
                           clip_on=False,
                           **kwargs)
    a = np.arange(256).reshape(1,256)/256.
    bbox_image.set_data(a)
    ax.add_artist(bbox_image)
ax = plt.subplot(122)
a = np.linspace(0, 1, 256).reshape(1,-1)
a = np.vstack((a,a))

maps = sorted(m for m in plt.cm.datad if not m.endswith("_r"))

ncol = 2
nrow = len(maps) // ncol + 1

xpad_fraction = 0.3
dx = 1./(ncol + xpad_fraction*(ncol-1))

ypad_fraction = 0.3
dy = 1./(nrow + ypad_fraction*(nrow-1))

for i,m in enumerate(maps):
    ix, iy = divmod(i, nrow)
    bbox0 = Bbox.from_bounds(ix*dx*(1+xpad_fraction),
        1.-iy*dy*(1+ypad_fraction) - dy,
        dx, dy)
    bbox = TransformedBbox(bbox0, ax.transAxes)
    bbox_image = BboxImage(bbox,
        cmap = plt.get_cmap(m),
        norm = None,
        origin=None,
        **kwargs
    )
    bbox_image.set_data(a)
    ax.add_artist(bbox_image)

plt.draw()
plt.show()
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.image import BboxImage
from matplotlib._png import read_png
import matplotlib.colors
from matplotlib.cbook import get_sample_data

class RibbonBox(object):
    original_image = read_png(get_sample_data("Minduka_Present_Blue_Pack.png", asfileobj=False))
    cut_location = 70
    b_and_h = original_image[:,:,2]
    color = original_image[:,:,2] - original_image[:,:,0]
    alpha = original_image[:,:,3]
    nx = original_image.shape[1]

    def __init__(self, color):
        rgb = matplotlib.colors.colorConverter.to_rgb(color)

        im = np.empty(self.original_image.shape,
im[:, :, :3] = self.b_and_h[:, :, np.newaxis]
im[:, :, :3] -= self.color[:, :, np.newaxis]*(1.-np.array(rgb))
im[:, :, 3] = self.alpha

self.im = im

def get_stretched_image(self, stretch_factor):
    stretch_factor = max(stretch_factor, 1)
    ny, nx, nch = self.im.shape
    ny2 = int(ny*stretch_factor)
    stretched_image = np.empty((ny2, nx, nch), np.dtype(self.im.dtype))
    cut = self.im[self.cut_location,:, :]
    stretched_image[:, :, :] = cut
    stretched_image[:, self.cut_location,:, :] = \
        self.im[:, self.cut_location,:, :]
    stretched_image[-(ny-self.cut_location):,:, :] = \
        self.im[-(ny-self.cut_location):,:, :]
    self._cached_im = stretched_image
    return stretched_image

class RibbonBoxImage(BboxImage):
    zorder = 1

    def __init__(self, bbox, color,
                 cmap = None,
                 norm = None,
                 interpolation=None,
                 origin=None,
                 filternorm=1,
                 filterrad=4.0,
                 resample = False,
                 **kwargs):
        BboxImage.__init__(self, bbox,
                           cmap = cmap,
                           norm = norm,
                           interpolation=interpolation,
                           origin=origin,
                           filternorm=filternorm,
                           filterrad=filterrad,
                           resample = resample,
                           **kwargs)
```python
self._ribbonbox = RibbonBox(color)
self._cached_ny = None

def draw(self, renderer, *args, **kwargs):
    bbox = self.get_window_extent(renderer)
    stretch_factor = bbox.height / bbox.width
    ny = int(stretch_factor * self._ribbonbox.nx)
    if self._cached_ny != ny:
        arr = self._ribbonbox.get_stretched_image(stretch_factor)
        self.set_array(arr)
        self._cached_ny = ny
    BboxImage.draw(self, renderer, *args, **kwargs)

if 1:
    from matplotlib.transforms import Bbox, TransformedBbox
    from matplotlib.ticker import ScalarFormatter

fig, ax = plt.subplots()
years = np.arange(2004, 2009)
box_colors = [(0.8, 0.2, 0.2),
              (0.2, 0.8, 0.2),
              (0.2, 0.2, 0.8),
              (0.7, 0.5, 0.8),
              (0.3, 0.8, 0.7),
              ]
heights = np.random.random(years.shape) * 7000 + 3000
fmt = ScalarFormatter(useOffset=False)
ax.xaxis.set_major_formatter(fmt)
for year, h, bc in zip(years, heights, box_colors):
    bbox0 = Bbox.from_extents(year-0.4, 0., year+0.4, h)
    bbox = TransformedBbox(bbox0, ax.transData)
    rb_patch = RibbonBoxImage(bbox, bc, interpolation="bicubic")
    ax.add_artist(rb_patch)
    ax.annotate(r"%d" % (int(h/100.)*100),
                (year, h), va="bottom", ha="center")

patch_gradient = BboxImage(ax.bbox,
                            interpolation="bicubic",
                            zorder=0.1,
                            )
gradient = np.zeros((2, 2, 4), dtype=np.float)
gradient[:,:,3] = [0.1, 0.3,
                   [0.3, 0.5]
               ] # alpha channel
```

84.61. pylab_examples example code: demo_ribbon_box.py
patch_gradient.set_array(gradient)
ax.add_artist(patch_gradient)

ax.set_xlim(years[0]-0.5, years[-1]+0.5)
ax.set_ylim(0, 10000)

fig.savefig('ribbon_box.png')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.62 pylab_examples example code: demo_text_path.py

```python
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from matplotlib.image import BboxImage
import numpy as np
from matplotlib.transforms import IdentityTransform
import matplotlib.patches as mpatches

# Example code for demo_text_path.py
```
from matplotlib.offsetbox import AnnotationBbox, AnchoredOffsetbox, AuxTransformBox
from matplotlib.cbook import get_sample_data
from matplotlib.text import TextPath

class PathClippedImagePatch(mpatches.PathPatch):
    ""
    The given image is used to draw the face of the patch. Internally, it uses BboxImage whose clippath set to the path of the patch.
    ""
    def __init__(self, path, bbox_image, **kwargs):
        mpatches.PathPatch.__init__(self, path, **kwargs)
        self._init_bbox_image(bbox_image)

def set_facecolor(self, color):
    """simply ignore facecolor""
    mpatches.PathPatch.set_facecolor(self, "none")

def _init_bbox_image(self, im):
    bbox_image = BboxImage(self.get_window_extent,
                           norm = None,
                           origin=None,
                          )
    bbox_image.set_transform(IdentityTransform())
    bbox_image.set_data(im)
    self.bbox_image = bbox_image

def draw(self, renderer=None):
    # the clip path must be updated every draw. any solution? -JJ
    self.bbox_image.set_clip_path(self._path, self.get_transform())
    self.bbox_image.draw(renderer)

    mpatches.PathPatch.draw(self, renderer)

if 1:
    usetex = plt.rcParams["text.usetex"]
    fig = plt.figure(1)

    # EXAMPLE 1
    ax = plt.subplot(211)
```python
from matplotlib._png import read_png
fn = get_sample_data("grace_hopper.png", asfileobj=False)
arr = read_png(fn)

text_path = TextPath((0, 0), "!?", size=150)
p = PathClippedImagePatch(text_path, arr, ec="k",
    transform=IdentityTransform())

#p.set_clip_on(False)

# make offset box
offsetbox = AuxTransformBox(IdentityTransform())
offsetbox.add_artist(p)

# make anchored offset box
ao = AnchoredOffsetbox(loc=2, child=offsetbox, frameon=True, borderpad=0.2)
ax.add_artist(ao)

# another text
from matplotlib.patches import PathPatch
if usetex:
    r = r"\mbox{textpath supports mathtext \& \TeX}"
else:
    r = r"textpath supports mathtext & \TeX"

text_path = TextPath((0, 0), r,
    size=20, usetex=usetex)

p1 = PathPatch(text_path, ec="w", lw=3, fc="w", alpha=0.9,
    transform=IdentityTransform())
p2 = PathPatch(text_path, ec="none", fc="k",
    transform=IdentityTransform())

offsetbox2 = AuxTransformBox(IdentityTransform())
offsetbox2.add_artist(p1)
offsetbox2.add_artist(p2)

ab = AnnotationBbox(offsetbox2, (0.95, 0.05),
    xycoords='axes fraction',
    boxcoords="offset points",
    box_alignment=(1.,0.),
    frameon=False)

ax.add_artist(ab)

ax.imshow([[0,1,2],[1,2,3]], cmap=plt.cm.gist_gray_r,
    interpolation="bilinear",
    aspect="auto")
```

# EXAMPLE 2
ax = plt.subplot(212)

arr = np.arange(256).reshape(1,256)/256.

if usetex:
    s = r"\displaystyle\left[\sum_{n=1}^{\infty}\frac{-e^{i\pi}}{2^n}\right]!"
else:
    s = r"\left[\sum_{n=1}^{\infty}\frac{-e^{i\pi}}{2^n}\right]!"

text_path = TextPath((0, 0), s, size=40, usetex=usetex)
text_patch = PathClippedImagePatch(text_path, arr, ec="none",
                                    transform=IdentityTransform())

shadow1 = mpatches.Shadow(text_patch, 1, -1, props=dict(fc="none", ec="0.6", lw=3))
shadow2 = mpatches.Shadow(text_patch, 1, -1, props=dict(fc="0.3", ec="none"))

# make offset box
offsetbox = AuxTransformBox(IdentityTransform())
offsetbox.add_artist(shadow1)
offsetbox.add_artist(shadow2)
offsetbox.add_artist(text_patch)

# place the anchored offset box using AnnotationBbox
ab = AnnotationBbox(offsetbox, (0.5, 0.5),
                     xycoords='data',
                     boxcoords="offset points",
                     box_alignment=(0.5,0.5),
                     )

#text_path.set_size(10)

ax.add_artist(ab)

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#clf()
from mpl_toolkits.axes_grid1.axes_grid import ImageGrid
def test_rotation_mode(fig, mode, subplot_location):
    ha_list = "left center right".split()
    va_list = "top center baseline bottom".split()
    grid = ImageGrid(fig, subplot_location,
                      nrows_ncols=(len(va_list), len(ha_list)),
                      share_all=True, aspect=True, #label_mode='1',
                      cbar_mode=None)

    for ha, ax in zip(ha_list, grid.axes_row[-1]):
        ax.axis["bottom"].label.set_text(ha)

    grid.axes_row[0][1].set_title(mode, size="large")

    for va, ax in zip(va_list, grid.axes_column[0]):
        ax.axis["left"].label.set_text(va)

    i = 0
    for va in va_list:
        for ha in ha_list:
            ax = grid[i]
            for axis in ax.axis.values():
                axis.toggle(ticks=False, ticklabels=False)
ax.text(0.5, 0.5, "Tpg",
        size="large", rotation=40,
        bbox=dict(boxstyle="square,pad=0.",
                  ec="none", fc="0.5", alpha=0.5),
        ha=ha, va=va,
        rotation_mode=mode)
ax.axvline(0.5)
ax.axhline(0.5)
i += 1

if 1:
    import matplotlib.pyplot as plt
    fig = plt.figure(1, figsize=(5.5,4))
    fig.clf()
    test_rotation_mode(fig, "default", 121)
    test_rotation_mode(fig, "anchor", 122)
    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.64  *pylab_examples* example code: demo_tight_layout.py

![Graph](image-url)
Chapter 84. pylab_examples Examples
Title

y-label

x-label

Title

y-label

x-label
2070 Chapter 84. pylab_examples Examples
```python
import matplotlib.pyplot as plt
import random

fontsizes = [8, 16, 24, 32]
def example_plot(ax):
    ax.plot([1, 2])
    ax.set_xlabel('x-label', fontsize=random.choice(fontsizes))
    ax.set_ylabel('y-label', fontsize=random.choice(fontsizes))
    ax.set_title('Title', fontsize=random.choice(fontsizes))

fig, ax = plt.subplots()
example_plot(ax)
plt.tight_layout()

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()

fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1)
example_plot(ax1)
example_plot(ax2)
```

import matplotlib.pyplot as plt

import random

fontsizes = [8, 16, 24, 32]
def example_plot(ax):
    ax.plot([1, 2])
    ax.set_xlabel('x-label', fontsize=random.choice(fontsizes))
    ax.set_ylabel('y-label', fontsize=random.choice(fontsizes))
    ax.set_title('Title', fontsize=random.choice(fontsizes))

fig, ax = plt.subplots()
example_plot(ax)
plt.tight_layout()

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()

fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1)
example_plot(ax1)
example_plot(ax2)
plt.tight_layout()

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
example_plot(ax1)
example_plot(ax2)
plt.tight_layout()

fig, axes = plt.subplots(nrows=3, ncols=3)
for row in axes:
    for ax in row:
        example_plot(ax)
plt.tight_layout()

fig = plt.figure()
ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
plt.tight_layout()

fig = plt.figure()
ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()

plt.show()

fig = plt.figure()
import matplotlib.gridspec as gridspec
gs1 = gridspec.GridSpec(3, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])
ax3 = fig.add_subplot(gs1[2])
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

gs1.tight_layout(fig, rect=[None, None, 0.45, None])

gs2 = gridspec.GridSpec(2, 1)
ax4 = fig.add_subplot(gs2[0])
ax5 = fig.add_subplot(gs2[1])

#example_plot(ax4)
#example_plot(ax5)

gs2.tight_layout(fig, rect=[0.45, None, None, None])

# now match the top and bottom of two gridspecs.
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, PathPatch
from matplotlib.path import Path
from matplotlib.transforms import Affine2D
import numpy as np

r = np.random.rand(50)
t = np.random.rand(50) * np.pi * 2.0
x = r * np.cos(t)
y = r * np.sin(t)
```
fig, ax = plt.subplots(figsize=(6,6))
circle = Circle((0, 0), 1, facecolor='none',
    edgecolor=(0,0.8,0.8), linewidth=3, alpha=0.5)
ax.add_patch(circle)

im = plt.imshow(np.random.random((100, 100)),
    origin='lower', cmap=cm.winter,
    interpolation='spline36',
    extent=[-1, 1, -1, 1])
im.set_clip_path(circle)

plt.plot(x, y, 'o', color=(0.9, 0.9, 1.0), alpha=0.8)

# Dolphin from OpenClipart library by Andy Fitzsimon
# <cc:License rdf:about="http://web.resource.org/cc/PublicDomain">
#   <cc:permits rdf:resource="http://web.resource.org/cc/Reproduction"/>
#   <cc:permits rdf:resource="http://web.resource.org/cc/Distribution"/>
# </cc:License>

dolphin = """
M -0.59739425,160.18173 C -0.62740401,160.18885 -0.57867129,160.11183
-0.57867129,160.11183 C -0.5438361,159.89315
-0.39514638,159.81496 C -0.24645668,159.73678 -0.18316813,159.71981
-0.18316813,159.71981 C -0.18316813,159.71981 -0.10322971,159.58124
-0.057804323,159.58725 C -0.029723983,159.58913 -0.06184103,159.60356
-0.07126513,159.62815 C -0.00250183,159.65325 -0.02918513,159.70554
-0.061841203,159.71248 C -0.040763903,159.7194 -0.0066711426,159.7191
0.077336307,159.73612 C 0.1689567,159.76377 0.2383036,159.83448
0.31516668,159.91533 C 0.3465303,159.96618 0.501127,160.177
0.501127,160.1771 C 0.63668998,160.19238 0.67763022,160.31259
0.66556395,160.32668 C 0.65339985,160.34212 0.66350443,160.33642
0.64907998,160.33088 C 0.63463742,160.32533 0.61309688,160.297
0.5789627,160.29339 C 0.54348657,160.28968 0.52329693,160.27674
0.50728856,160.27737 C 0.49060916,160.27795 0.48965803,160.31565
0.46114204,160.33673 C 0.43329696,160.35786 0.4570711,160.39871
0.43309565,160.40685 C 0.4105108,160.41442 0.39416631,160.33027
0.3954995,160.2933 C 0.39683269,160.25672 0.43807996,160.21522
0.44567915,160.19734 C 0.45327833,160.17946 0.27946869,159.9424
-0.061852613,159.99845 C -0.083965233,160.0427 -0.26176109,160.06683
-0.26176109,160.06683 C -0.30127925,160.07028 -0.21167141,160.09731
-0.24649368,160.1011 C -0.32642366,160.11569 -0.34521187,160.06895
-0.466222933,160.0819 C -0.4672344,160.09485 -0.5673844,160.17461
-0.59739425,160.18173
"""
vertices = []
codes = []
parts = dolphin.split()
i = 0
code_map = {
    'M': (Path.MOVETO, 1),
    'C': (Path.CURVE4, 3),
'L': (Path.LINETO, 1)
}

while i < len(parts):
    code = parts[i]
    path_code, npoints = code_map[code]
    codes.extend([path_code] * npoints)
    vertices.extend([[float(x) for x in y.split(',')] for y in parts[i+1:i+npoints+1]])
    i += npoints + 1
vertices = np.array(vertices, np.float)
vertices[:,1] -= 160

dolphin_path = Path(vertices, codes)
dolphin_patch = PathPatch(dolphin_path, facecolor=(0.6, 0.6, 0.6),
                           edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphin_patch)

vertices = Affine2D().rotate_deg(60).transform(vertices)
dolphin_path2 = Path(vertices, codes)
dolphin_patch2 = PathPatch(dolphin_path2, facecolor=(0.5, 0.5, 0.5),
                           edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphin_patch2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.collections import EllipseCollection

x = np.arange(10)
y = np.arange(15)
X, Y = np.meshgrid(x, y)
XY = np.hstack((X.ravel()[:, np.newaxis], Y.ravel()[:, np.newaxis]))

ww = X/10.0
hh = Y/15.0
aa = X*9

fig, ax = plt.subplots()
ec = EllipseCollection(ww, hh, aa, units='x',

```

84.66  pylab_examples example code: ellipse_collection.py
offsets=XY,
    transOffset=ax.transData)
ec.set_array((X+Y).ravel())
ax.add_collection(ec)
ax.autoscale_view()
ax.set_xlabel('X')
ax.set_ylabel('y')
cbar = plt.colorbar(ec)
cbar.set_label('X+Y')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.67  pylab_examples example code: ellipse_demo.py

from pylab import figure, show, rand
from matplotlib.patches import Ellipse

NUM = 250

ells = [Ellipse(xy=rand(2)*10, width=rand(), height=rand(), angle=rand()*360)
    for i in range(NUM)]
fig = figure()
ax = fig.add_subplot(111, aspect='equal')
for e in ells:
    ax.add_artist(e)
    e.set_clip_box(ax.bbox)
    e.set_alpha(rand())
    e.set_facecolor(rand(3))

ax.set_xlim(0, 10)
ax.set_ylim(0, 10)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.68  pylab_examples example code: ellipse_rotated.py

from pylab import *
from matplotlib.patches import Ellipse

delta = 45.0  # degrees
```python
import matplotlib.pyplot as plt
import numpy as np

angles = np.arange(0, 360+delta, delta)
ells = [Ellipse((1, 1), 4, 2, a) for a in angles]

a = plt.subplot(111, aspect='equal')

for e in ells:
    e.set_clip_box(a.bbox)
    e.set_alpha(0.1)
    a.add_artist(e)

plt.xlim(-2, 4)
plt.ylim(-1, 3)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

---

84.69  pylab_examples example code: equal_aspect_ratio.py

```bash
#!/usr/bin/env python

""
Example: simple line plot.

#about as simple as it gets folks
```
Show how to make a plot that has equal aspect ratio

```python
from pylab import *

t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*2*pi*t)
plot(t, s, '-', lw=2)
xlabel('time (s)')
ylabel('voltage (mV)')
title('About as simple as it gets, folks')
grid(True)

axes().set_aspect('equal', 'datalim')

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.70 pylab_examples example code: errorbar_limits.py
Illustration of upper and lower limit symbols on errorbars

```python
from math import pi
from numpy import array, arange, sin
import pylab as P

fig = P.figure()
x = arange(10.0)
y = sin(arange(10.0)/20.0*pi)
P.errorbar(x,y,yerr=0.1,capsize=3)

y = sin(arange(10.0)/20.0*pi) + 1
P.errorbar(x,y,yerr=0.1, uplims=True)

y = sin(arange(10.0)/20.0*pi) + 2
upperlimits = array([1,0]*5)
lowerlimits = array([0,1]*5)
P.errorbar(x, y, yerr=0.1, uplims=upperlimits, lolims=lowerlimits)

P.xlim(-1,10)
fig = P.figure()
```

84.70. pylab_examples example code: errorbar_limits.py
x = arange(10.0)/10.0
y = (x+0.1)**2
P.errorbar(x, y, xerr=0.1, xlolims=True)
y = (x+0.1)**3
P.errorbar(x+0.6, y, xerr=0.1, xuplims=upperlimits, xlolims=lowerlimits)
y = (x+0.1)**4
P.errorbar(x+1.2, y, xerr=0.1, xuplims=True)
P.xlim(-0.2,2.4)
P.ylim(-0.1,1.3)
P.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.71  pylab_examples example code: errorbar_subsample.py

```
Demo for the errorevery keyword to show data full accuracy data plots with
```
few errorbars.

```python
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.1)
y = np.exp(-x)

# example variable error bar values
yerr = 0.1 + 0.1*np.sqrt(x)

# Now switch to a more OO interface to exercise more features.
fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True)
ax = axs[0]
ax.errorbar(x, y, yerr=yerr)
ax.set_title('all errorbars')

ax = axs[1]
ax.errorbar(x, y, yerr=yerr, errorevery=5)
ax.set_title('only every 5th errorbar')

fig.suptitle('Errorbar subsampling for better visualibility')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
# -*- Coding:utf-8 -*-
"Plot two curves, then use EventCollections to mark the locations of the x and y data points on the respective axes for each curve"

import matplotlib.pyplot as plt
from matplotlib.collections import EventCollection
import numpy as np

# create random data
np.random.seed(50)
xdata = np.random.random([2, 10])

# split the data into two parts
xdata1 = xdata[0, :]
xdata2 = xdata[1, :]

# sort the data so it makes clean curves
xdata1.sort()
xdata2.sort()

# create some y data points
ydata1 = xdata1 ** 2
ydata2 = 1 - xdata2 ** 3

# plot the data
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(xdata1, ydata1, 'r', xdata2, ydata2, 'b')

# create the events marking the x data points
xevents1 = EventCollection(xdata1, color=[1, 0, 0], linelength=0.05)
xevents2 = EventCollection(xdata2, color=[0, 0, 1], linelength=0.05)

# create the events marking the y data points
yevents1 = EventCollection(ydata1, color=[1, 0, 0], linelength=0.05, orientation='vertical')
yevents2 = EventCollection(ydata2, color=[0, 0, 1], linelength=0.05, orientation='vertical')

# add the events to the axis
ax.add_collection(xevents1)
ax.add_collection(xevents2)
ax.add_collection(yevents1)
ax.add_collection(yevents2)

# set the limits
ax.set_xlim([0, 1])
ax.set_ylim([0, 1])

ax.set_title('line plot with data points')

# display the plot
plt.show()
#!/usr/bin/env python
# -*- Coding:utf-8 -*-

"an eventplot showing sequences of events with various line properties
the plot is shown in both horizontal and vertical orientations"

import matplotlib.pyplot as plt
import numpy as np
import matplotlib
matplotlib.rcParams['font.size'] = 8.0

# set the random seed
np.random.seed(0)

# create random data
data1 = np.random.random([6, 50])

# set different colors for each set of positions
colors1 = np.array([[1, 0, 0],
                     [0, 1, 0],
                     [0, 0, 1],
                     [1, 1, 0],
                     [1, 0, 1],
                     [0, 1, 1]])
# set different line properties for each set of positions
# note that some overlap
lineoffsets1 = np.array([-15, -3, 1, 1.5, 6, 10])
linelengths1 = [5, 2, 1, 1, 3, 1.5]

fig = plt.figure()

# create a horizontal plot
ax1 = fig.add_subplot(221)
ax1.eventplot(data1, colors=colors1, lineoffsets=lineoffsets1, linelengths=linelengths1)

# create a vertical plot
ax2 = fig.add_subplot(223)
ax2.eventplot(data1, colors=colors1, lineoffsets=lineoffsets1, linelengths=linelengths1, orientation='vertical')

# create another set of random data.
# the gamma distribution is only used fo aesthetic purposes
data2 = np.random.gamma(4, size=[60, 50])

# use individual values for the parameters this time
# these values will be used for all data sets (except lineoffsets2, which
# sets the increment between each data set in this usage)
colors2 = [[0, 0, 0]]
lineoffsets2 = 1
linelengths2 = 1

# create a horizontal plot
ax1 = fig.add_subplot(222)
ax1.eventplot(data2, colors=colors2, lineoffsets=lineoffsets2, linelengths=linelengths2)

# create a vertical plot
ax2 = fig.add_subplot(224)
ax2.eventplot(data2, colors=colors2, lineoffsets=lineoffsets2, linelengths=linelengths2, orientation='vertical')

plt.show()
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

styles = mpatches.ArrowStyle.get_styles()

ncol=2
nrow = (len(styles)+1) // ncol
figheight = (nrow*0.5)
fig1 = plt.figure(1, (4.*ncol/1.5, figheight/1.5))
fontsize = 0.2 * 70

ax = fig1.add_axes([0, 0, 1, 1], frameon=False, aspect=1.)
ax.set_xlim(0, 4*ncol)
ax.set_ylim(0, figheight)

def to_texstring(s):
    s = s.replace("<", r"$<$")

import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

styles = mpatches.ArrowStyle.get_styles()

ncol=2
nrow = (len(styles)+1) // ncol
figheight = (nrow*0.5)
fig1 = plt.figure(1, (4.*ncol/1.5, figheight/1.5))
fontsize = 0.2 * 70

ax = fig1.add_axes([0, 0, 1, 1], frameon=False, aspect=1.)
ax.set_xlim(0, 4*ncol)
ax.set_ylim(0, figheight)

def to_texstring(s):
    s = s.replace("<", r"$<$")
```python
s = s.replace(">", r"$>$")
s = s.replace('|', r"$|$")
return s

for i, (stylename, styleclass) in enumerate(sorted(styles.items())):
x = 3.2 + (i // nrow) * 4
y = (figheight - 0.7 - i % nrow) / figheight
p = mpatches.Circle((x, y), 0.2, fc="w")
ax.add_patch(p)

ax.annotate(to_texstring(stylename), (x, y),
            (x - 1.2, y),
            xycoords="figure fraction", textcoords="figure fraction",
            ha="right", va="center",
            size=fontsize,
            arrowprops=dict(arrowstyle=stylename,
                            patchB=p,
                            shrinkA=5,
                            shrinkB=5,
                            fc="w", ec="k",
                            connectionstyle="arc3, rad=-0.05",
                            )
            ,
            bbox=dict(boxstyle="square", fc="w"))

ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
from matplotlib.patches import FancyBboxPatch

# Bbox object around which the fancy box will be drawn.
bb = mtransforms.Bbox([[0.3, 0.4], [0.7, 0.6]])

def draw_bbox(ax, bb):
    # boxstyle=square with pad=0, i.e. bbox itself.
    p_bbox = FancyBboxPatch((bb.xmin, bb.ymin),
        abs(bb.width), abs(bb.height),
        boxstyle="square,pad=0.",
        ec="k", fc="none", zorder=10.,
    )

    ax.add_patch(p_bbox)

def test1(ax):
    # a fancy box with round corners. pad=0.1
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
        abs(bb.width), abs(bb.height),
        boxstyle="round,pad=0.1",
        rounding_size=0.2,
        mutation_scale=2,
    )

    ax.add_patch(p_fancy)
boxstyle="round, pad=0.1",
fc=(1., .8, 1.),
ec=(1., 0.5, 1.))

ax.add_patch(p_fancy)

ax.text(0.1, 0.8,
    r'boxstyle="round, pad=0.1"
    size=10, transform=ax.transAxes)

# draws control points for the fancy box.
# l = p_fancy.get_path().vertices
# ax.plot(l[:,0], l[:,1], ".")

def test2(ax):
    # bbox=round has two optional argument: pad and rounding_size. 
    # They can be set during the initialization.
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
        abs(bb.width), abs(bb.height),
        boxstyle="round, pad=0.1",
        fc=(1., .8, 1.),
        ec=(1., 0.5, 1.))

    ax.add_patch(p_fancy)

    # boxstyle and its argument can be later modified with 
    # set_boxstyle method. Note that the old attributes are simply 
    # forgotten even if the boxstyle name is same.
    p_fancy.set_boxstyle("round, pad=0.1, rounding_size=0.2")
    # or
    #p_fancy.set_boxstyle("round", pad=0.1, rounding_size=0.2)

    ax.text(0.1, 0.8,
        r' boxstyle="round, pad=0.1\n        rounding\_size=0.2"
        size=10, transform=ax.transAxes)

    # draws control points for the fancy box.
    # l = p_fancy.get_path().vertices
    # ax.plot(l[:,0], l[:,1], ".")

def test3(ax):

draw_bbox(ax, bb)
# mutation_scale determine overall scale of the mutation,  
# i.e. both pad and rounding_size is scaled according to this  
# value.
p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),  
    abs(bb.width), abs(bb.height),  
    boxstyle="round,pad=0.1",  
    mutation_scale=2.,  
    fc=(1., .8, 1.),  
    ec=(1., .5, 1.))

ax.add_patch(p_fancy)

ax.text(0.1, 0.8,  
    'boxstyle="round,pad=0.1"
mutation\_scale=2',  
    size=10, transform=ax.transAxes)

# draws control points for the fancy box.
# l = p_fancy.get_path().vertices
# ax.plot(l[:,0], l[:,1], ".")
draw_bbox(ax, bb)

def test4(ax):

    # When the aspect ratio of the axes is not 1, the fancy box may  
    # not be what you expected (green)
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),  
        abs(bb.width), abs(bb.height),  
        boxstyle="round,pad=0.2",  
        fc="none",  
        ec=(0., .5, 0.), zorder=4)

    ax.add_patch(p_fancy)

    # You can compensate this by setting the mutation_aspect (pink).
p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),  
    abs(bb.width), abs(bb.height),  
    boxstyle="round,pad=0.3",  
    mutation_aspect=.5,  
    fc=(1., 0.8, 1.),  
    ec=(1., .5, 1.))

    ax.add_patch(p_fancy)

    ax.text(0.1, 0.8,  
        'boxstyle="round,pad=0.3"
mutation\_aspect=.5',  
        size=10, transform=ax.transAxes)
```python
def test_all():
    plt.clf()

    ax = plt.subplot(2, 2, 1)
    test1(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_title("test1")
    ax.set_aspect(1.)

    ax = plt.subplot(2, 2, 2)
    ax.set_title("test2")
    test2(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_aspect(1.)

    ax = plt.subplot(2, 2, 3)
    ax.set_title("test3")
    test3(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_aspect(1)

    ax = plt.subplot(2, 2, 4)
    ax.set_title("test4")
    test4(ax)
    ax.set_xlim(-0.5, 1.5)
    ax.set_ylim(0., 1.)
    ax.set_aspect(2.)

    plt.draw()
    plt.show()

test_all()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.patches as mpatch
import matplotlib.pyplot as plt

styles = mpatch.BoxStyle.get_styles()

figheight = (len(styles)+.5)
fig1 = plt.figure(1, (4/1.5, figheight/1.5))

for i, (stylename, styleclass) in enumerate(styles.items()):
    fig1.text(0.5, (float(len(styles)) - 0.5 - i)/figheight, stylename,
              ha="center",
              size=0.3 * 72,
              transform=fig1.transFigure,
              bbox=dict(boxstyle=stylename, fc="w", ec="k"))
plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.77 pylab_examples example code: fancytextbox_demo.py

```python
import matplotlib.pyplot as plt

plt.text(0.6, 0.5, "test", size=50, rotation=30.,
         ha="center", va="center",
         bbox = dict(boxstyle="round",
                     ec=(1., 0.5, 0.5),
                     fc=(1., 0.8, 0.8),
                    )
)

plt.text(0.5, 0.4, "test", size=50, rotation=-30.,
         ha="right", va="top",
         bbox = dict(boxstyle="square",
                     ec=(1., 0.5, 0.5),
                     fc=(1., 0.8, 0.8),
                    )
)
```

84.77. pylab_examples example code: fancytextbox_demo.py
import numpy as np
import matplotlib
import matplotlib.cm as cm
import matplotlib.pyplot as plt

fig = plt.figure()
```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.7])
ax2 = fig.add_axes([0.55, 0.1, 0.4, 0.7])

x = np.arange(0.0, 2.0, 0.02)
y1 = np.sin(2*np.pi*x)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
y2 = np.exp(-x)
l1, l2 = ax1.plot(x, y1, 'rs-', x, y2, 'go')

y3 = np.sin(4*np.pi*x)
y4 = np.exp(-2*x)
l3, l4 = ax2.plot(x, y3, 'yd-', x, y4, 'k^')

fig.legend((l1, l2), ('Line 1', 'Line 2'), 'upper left')
fig.legend((l3, l4), ('Line 3', 'Line 4'), 'upper right')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.80 pylab_examples example code: figure_title.py

```python
#!/usr/bin/env python
from matplotlib.font_manager import FontProperties
from pylab import *

def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
```

Chapter 84. pylab_examples Examples
return multiply(s1,e1)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

subplot(121)
plot(t1, f(t1), 'bo', t2, f(t2), 'k')
title('subplot 1')
ylabel('Damped oscillation')
suptitle('This is a somewhat long figure title', fontsize=16)

subplot(122)
plot(t3, cos(2*pi*t3), 'r--')
xlabel('time (s)')
title('subplot 2')
ylabel('Undamped')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.81 *pylab_examples* example code: `fill_between_demo.py`

![Graph showing fill_between for different y ranges]
fill between where

Now regions with y2>1 are masked
```python
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig, (ax1, ax2, ax3) = plt.subplots(3,1, sharex=True)
ax1.fill_between(x, 0, y1)
ax1.set_ylabel('between y1 and 0')
ax2.fill_between(x, y1, 1)
ax2.set_ylabel('between y1 and 1')
ax3.fill_between(x, y1, y2)
ax3.set_ylabel('between y1 and y2')
ax3.set_xlabel('x')

# now fill_between y1 and y2 where a logical condition is met. Note
# this is different than calling
# fill_between(x[where], y1[where],y2[where])
# because of edge effects over multiple contiguous regions.
fig, (ax, ax1) = plt.subplots(2, 1, sharex=True)
```

2106 Chapter 84. pylab_examples Examples
ax.plot(x, y1, x2, color='black')
ax.fill_between(x, y1, y2, where=\(y_2\geq y_1\), facecolor='green', interpolate=True)
ax.fill_between(x, y1, y2, where=\(y_2\leq y_1\), facecolor='red', interpolate=True)
ax.set_title('fill between where')

# Test support for masked arrays.
y2 = np.ma.masked_greater(y2, 1.0)
ax1.plot(x, y1, x2, color='black')
ax1.fill_between(x, y1, y2, where=\(y_2\geq y_1\), facecolor='green', interpolate=True)
ax1.fill_between(x, y1, y2, where=\(y_2\leq y_1\), facecolor='red', interpolate=True)
ax1.set_title('Now regions with \(y_2>1\) are masked')

# This example illustrates a problem; because of the data
# gridding, there are undesired unfilled triangles at the crossover
# points. A brute-force solution would be to interpolate all
# arrays to a very fine grid before plotting.

# show how to use transforms to create axes spans where a certain condition is satisfied
fig, ax = plt.subplots()
y = np.sin(\(4\pi x\))
ax.plot(x, y, color='black')

# use the data coordinates for the x-axis and the axes coordinates for the y-axis
import matplotlib.transforms as mtransforms
trans = mtransforms.blended_transform_factory(ax.transData, ax.transAxes)
theta = 0.9
ax.axhline(theta, color='green', lw=2, alpha=0.5)
ax.axhline(-theta, color='red', lw=2, alpha=0.5)
ax.fill_between(x, 0, 1, where=y>theta, facecolor='green', alpha=0.5, transform=trans)
ax.fill_between(x, 0, 1, where=y<-theta, facecolor='red', alpha=0.5, transform=trans)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.82 \texttt{pylab_examples} example code: fill\_betweenx\_demo.py

![Plot showing the use of `fill_betweenx` function to fill between curves in y direction.](image)
import matplotlib.mlab as mlab
from matplotlib.pyplot import figure, show
import numpy as np

## Copy of fill_between.py but using fill_betweenx() instead.

x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig = figure()
x1 = fig.add_subplot(311)
x2 = fig.add_subplot(312, sharex=ax1)
x3 = fig.add_subplot(313, sharex=ax1)

ax1.fill_betweenx(x, 0, y1)
ax1.set_ylabel('between y1 and 0')

ax2.fill_betweenx(x, y1, 1)
ax2.set_ylabel('between y1 and 1')

ax3.fill_betweenx(x, y1, y2)
ax3.set_ylabel('between y1 and y2')
ax3.set_xlabel('x')
# now fill between y1 and y2 where a logical condition is met. Note
# this is different than calling
# fill_between(x[where], y1[where], y2[where])
# because of edge effects over multiple contiguous regions.
fig = figure()
ax = fig.add_subplot(2, 1, 1)
an.plot(y1, x, x, y2, x, color='black')
ax.fill_betweenx(x, y1, y2, where=y2 >= y1, facecolor='green')
ax.fill_betweenx(x, y1, y2, where=y2 <= y1, facecolor='red')
ax.set_title('fill between where')

# Test support for masked arrays.
y2 = np.ma.masked_greater(y2, 1.0)
ax1 = fig.add_subplot(2, 1, 2, sharex=ax)
an1.plot(y1, x, x, y2, x, color='black')
ax1.fill_betweenx(x, y1, y2, where=y2 >= y1, facecolor='green')
ax1.fill_betweenx(x, y1, y2, where=y2 <= y1, facecolor='red')
ax1.set_title('Now regions with y2 > 1 are masked')

# This example illustrates a problem; because of the data
# gridding, there are undesired unfilled triangles at the crossover
# points. A brute-force solution would be to interpolate all
# arrays to a very fine grid before plotting.
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

from pylab import *

theta = arange(0,8*pi,0.1)
a=1
b=.2

for dt in arange(0,2*pi,pi/2.0):
    x = a*cos( theta+dt )*exp(b*theta)
    y = a*sin( theta+dt )*exp(b*theta)
    dt = dt+pi/4.0
    x2 = a*cos( theta+dt )*exp(b*theta)
    y2 = a*sin( theta+dt )*exp(b*theta)
    xf = concatenate( (x,x2[::-1]) )
    yf = concatenate( (y,y2[::-1]) )
    p1=fill(xf,yf)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.84 pylab_examples example code: finance_demo.py

```python
#!/usr/bin/env python
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter, WeekdayLocator,
    DayLocator, MONDAY
from matplotlib.finance import quotes_historical_yahoo_ohlc, candlestick_ohlc

# (Year, month, day) tuples suffice as args for quotes_historical_yahoo
date1 = (2004, 2, 1)
date2 = (2004, 4, 12)

mondays = WeekdayLocator(MONDAY)       # major ticks on the mondays
alldays = DayLocator()                 # minor ticks on the days
weekFormatter = DateFormatter('%b %d') # e.g., Jan 12
dayFormatter = DateFormatter('%d')     # e.g., 12

plt.figure(num=1, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
plt.title('finance demo')
plt.xlabel('Date')
plt.ylabel('Price')
plt.grid(True)
plt.xticks(mondays, weekFormatter.formatdate(date) for date in alldays)
plt.yticks(alldays, dayFormatter.formatdate(date) for date in alldays)
plt.plot_date(date, price, 'r-')
plt.show()
```

2112 Chapter 84. pylab_examples Examples
quotes = quotes_historical_yahoo_ohlc('INTC', date1, date2)
if len(quotes) == 0:
    raise SystemExit

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2)
ax.xaxis.set_major_locator(mondays)
ax.xaxis.set_minor_locator(alldays)
ax.xaxis.set_major_formatter(weekFormatter)
#ax.xaxis.set_minor_formatter(dayFormatter)

#plot_day_summary(ax, quotes, ticksize=3)
candlestick_ohlc(ax, quotes, width=0.6)

ax.xaxis_date()
ax.autoscale_view()
plt.setp(plt.gca().get_xticklabels(), rotation=45, horizontalalignment='right')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import datetime
import numpy as np
import matplotlib.colors as colors
import matplotlib.finance as finance
import matplotlib.dates as mdates
import matplotlib.ticker as mticker
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import matplotlib.font_manager as font_manager

startdate = datetime.date(2006,1,1)
today = enddate = datetime.date.today()
ticker = 'SPY'

fh = finance.fetch_historical_yahoo(ticker, startdate, enddate)
# a numpy record array with fields: date, open, high, low, close, volume, adj_close)

r = mlab.csv2rec(fh); fh.close()
r.sort()
```python
def moving_average(x, n, type='simple'):
    """
    compute an n period moving average.

    type is 'simple' / 'exponential'
    """
    x = np.asarray(x)
    if type=='simple':
        weights = np.ones(n)
    else:
        weights = np.exp(np.linspace(-1., 0., n))
    weights /= weights.sum()
    a = np.convolve(x, weights, mode='full')[:len(x)]
a[:n] = a[n]
    return a

def relative_strength(prices, n=14):
    """
    compute the n period relative strength indicator
    http://stockcharts.com/school/doku.php?id=chart_school:glossary_r# relativestrengthindex
    http://www.investopedia.com/terms/r/rsi.asp
    """
    deltas = np.diff(prices)
    seed = deltas[:n+1]
    up = seed[seed>=0].sum()/n
    down = -seed[seed<0].sum()/n
    rs = up/down
    rsi = np.zeros_like(prices)
    rsi[:n] = 100. - 100./(1.+rs)
    for i in range(n, len(prices)):
        delta = deltas[i-1] # cause the diff is 1 shorter
        if delta>0:
            upval = delta
            downval = 0.
        else:
            upval = 0.
            downval = -delta
        up = (up**(n-1) + upval)/n
        down = (down**(n-1) + downval)/n
        rs = up/down
        rsi[i] = 100. - 100./(1.+rs)
    return rsi
```

def moving_average_convergence(x, nslow=26, nfast=12):
    
    compute the MACD (Moving Average Convergence/Divergence) using a fast and slow exponential moving avg
    return value is emaslow, emafast, macd which are len(x) arrays
    
    emaslow = moving_average(x, nslow, type='exponential')
    emafast = moving_average(x, nfast, type='exponential')
    return emaslow, emafast, emafast - emaslow

plt.rc('axes', grid=True)
plt.rc('grid', color='0.75', linestyle='-', linewidth=0.5)
textsize = 9
left, width = 0.1, 0.8
rect1 = [left, 0.7, width, 0.2]
rect2 = [left, 0.3, width, 0.4]
rect3 = [left, 0.1, width, 0.2]

fig = plt.figure(facecolor='white')
axescolor = '#f6f6f6'  # the axes background color
ax1 = fig.add_axes(rect1, axisbg=axescolor)  # left, bottom, width, height
ax2 = fig.add_axes(rect2, axisbg=axescolor, sharex=ax1)
ax2t = ax2.twinx()
ax3 = fig.add_axes(rect3, axisbg=axescolor, sharex=ax1)

### plot the relative strength indicator
prices = r.adj_close
rsi = relative_strength(prices)
fillcolor = 'darkgoldenrod'

ax1.plot(r.date, rsi, color=fillcolor)
ax1.axhline(70, color=fillcolor)
ax1.axhline(30, color=fillcolor)
ax1.fill_between(r.date, rsi, 70, where=(rsi>=70), facecolor=fillcolor, edgecolor=fillcolor)
ax1.fill_between(r.date, rsi, 30, where=(rsi<=30), facecolor=fillcolor, edgecolor=fillcolor)
ax1.text(0.6, 0.9, '>70 = overbought', va='top', transform=ax1.transAxes, fontsize=textsize)
ax1.text(0.6, 0.1, '<30 = oversold', transform=ax1.transAxes, fontsize=textsize)
ax1.set_ylim(0, 100)
ax1.set_yticks([30,70])
ax1.text(0.025, 0.95, 'RSI (14)', va='top', transform=ax1.transAxes, fontsize=textsize)
ax1.set_title('%s daily' %ticker)

### plot the price and volume data
dx = r.adj_close - r.close
low = r.low + dx
high = r.high + dx
deltas = np.zeros_like(prices)
deltas[1:] = np.diff(prices)
up = deltas>0
ax2.vlines(r.date[up], low[up], high[up], color='black', label='_nolegend_')
ax2.vlines(r.date[~up], low[~up], high[~up], color='black', label='_nolegend_')

ma20 = moving_average(prices, 20, type='simple')
ma200 = moving_average(prices, 200, type='simple')

linema20, = ax2.plot(r.date, ma20, color='blue', lw=2, label='MA (20)')
linema200, = ax2.plot(r.date, ma200, color='red', lw=2, label='MA (200)')

last = r[-1]
s = '%s O:%1.2f H:%1.2f L:%1.2f C:%1.2f, V:%1.1f M Chg:+%1.2f' % ( 
    today.strftime('%d-%b-%Y'), 
    last.open, last.high, 
    last.low, last.close, 
    last.volume*1e-6, 
    last.close-last.open )
t4 = ax2.text(0.3, 0.9, s, transform=ax2.transAxes, fontsize=textsize)

volume = (r.close*r.volume)/1e6  # dollar volume in millions
vmax = volume.max()

poly = ax2t.fill_between(r.date, volume, 0, label='Volume', facecolor=fillcolor, edgecolor=fillcolor)

### compute the MACD indicator

fillcolor = 'darkslategrey'
sslow = 26
nfast = 12
nema = 9
emaslow, emafast, macd = moving_average_convergence(prices, nslow=nslow, nfast=nfast)
ema9 = moving_average(macd, nema, type='exponential')

ax3.plot(r.date, macd, color='black', lw=2)
ax3.plot(r.date, ema9, color='blue', lw=1)

ax3.fill_between(r.date, macd-ema9, 0, alpha=0.5, facecolor=fillcolor, edgecolor=fillcolor)
ax3.text(0.025, 0.95, 'MACD (%d, %d, %d)'%(nfast, nslow, nema), va='top',
        transform=ax3.transAxes, fontsize=textsize)

# turn off upper axis tick labels, rotate the lower ones, etc
for ax in ax1, ax2, ax2t, ax3:
    if ax!=ax3:
        for label in ax.get_xticklabels():
            label.set_visible(False)
else:
    for label in ax.get_xticklabels():
        label.set_rotation(30)
        label.setHorizontalAlignment('right')

    ax.fmt_xdata = mdates.DateFormatter('%Y-%m-%d')

class MyLocator(mticker.MaxNLocator):
    def __init__(self, *args, **kwargs):
        mticker.MaxNLocator.__init__(self, *args, **kwargs)

    def __call__(self, *args, **kwargs):
        return mticker.MaxNLocator.__call__(self, *args, **kwargs)

# at most 5 ticks, pruning the upper and lower so they don't overlap
# with other ticks
#ax2.yaxis.set_major_locator(mticker.MaxNLocator(5, prune='both'))
#ax3.yaxis.set_major_locator(mticker.MaxNLocator(5, prune='both'))

ax2.yaxis.set_major_locator(MyLocator(5, prune='both'))
ax3.yaxis.set_major_locator(MyLocator(5, prune='both'))

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
# Recursively find all objects that match some criteria
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.text as text

a = np.arange(0,3,.02)
b = np.arange(0,3,.02)
c = np.exp(a)
d = c[::-1]

fig, ax = plt.subplots()
plt.plot(a,c,'
'k--',a,d,'
'k:',a,c+d,'
'k')
plt.legend(('Model length', 'Data length', 'Total message length'),
'upper center', shadow=True)
plt.ylim([-1,20])
plt.grid(False)
plt.xlabel('Model complexity --->')
plt.ylabel('Message length --->')
plt.title('Minimum Message Length')
```
# match on arbitrary function

```python
def myfunc(x):
    return hasattr(x, 'set_color') and not hasattr(x, 'set_facecolor')
```

```python
for o in fig.findobj(myfunc):
    o.set_color('blue')
```

# match on class instances

```python
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

```python
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 84.87 pylab_examples example code: font_table_ttf.py

```python
#!/usr/bin/env python
# -*- noplot -*-

""
matplotlib has support for freetype fonts. Here's a little example using the 'table' command to build a font table that shows the glyphs by character code.

Usage python font_table_ttf.py somefile.ttf
""

import sys
import os

import matplotlib
from matplotlib.ft2font import FT2Font
from matplotlib.font_manager import FontProperties
from pylab import figure, table, show, axis, title

import six
from six import unichr

# the font table grid

labelc = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F']
labelr = ['00', '10', '20', '30', '40', '50', '60', '70', '80', '90', 'A0', 'B0', 'C0', 'D0', 'E0', 'F0']

if len(sys.argv) > 1:
    fontname = sys.argv[1]
else:
    fontname = os.path.join(matplotlib.get_data_path(),
                        'fonts', 'ttf', 'Vera.ttf')

font = FT2Font(fontname)
codes = list(font.get_charmap().items())
codes.sort()

# a 16,16 array of character strings
chars = [['' for c in range(16)] for r in range(16)]
colors = [[(0.95, 0.95, 0.95) for c in range(16)] for r in range(16)]

figure(figsize=(8, 4), dpi=120)
for ccode, glyphind in codes:
    if ccode >= 256:
        continue
    r, c = divmod(ccode, 16)
    s = unichr(ccode)
    chars[r][c] = s
lightgrn = (0.5, 0.8, 0.5)
title(fontname)
tab = table(cellText=chars,
            rowLabels=labelr,
            colLabels=labelc,
            rowColours=[lightgrn]*16,
            colColours=[lightgrn]*16,
            cellColours=colors,
            cellLoc='center',
            loc='upper left')

for key, cell in tab.get_celld().items():
    row, col = key
    if row > 0 and col > 0:
        cell.set_text_props(fontproperties=FontProperties(fname=fontname))
axis('off')
show()
Show how to set custom font properties.

For interactive users, you can also use kwargs to the text command, which requires less typing. See examples/fonts_demo_kw.py

```python
#!/usr/bin/env python

from matplotlib.font_manager import FontProperties
from pylab import *

subplot(111, axisbg='w')

font0 = FontProperties()
alignment = {'horizontalalignment': 'center', 'verticalalignment': 'baseline'}

### Show family options

family = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']

font1 = font0.copy()
font1.set_size('large')

t = text(-0.8, 0.9, 'family', fontproperties=font1,
```
yp = [0.7, 0.5, 0.3, 0.1, -0.1, -0.3, -0.5]

for k in range(5):
    font = font0.copy()
    font.set_family(family[k])
    if k == 2:
        font.set_name('Script MT')
    t = text(-0.8, yp[k], family[k], fontproperties=font,
              **alignment)

### Show style options

style = ['normal', 'italic', 'oblique']

t = text(-0.4, 0.9, 'style', fontproperties=font1,
          **alignment)

for k in range(3):
    font = font0.copy()
    font.set_family('sans-serif')
    font.set_style(style[k])
    t = text(-0.4, yp[k], style[k], fontproperties=font,
             **alignment)

### Show variant options

variant = ['normal', 'small-caps']

for k in range(2):
    font = font0.copy()
    font.set_family('serif')
    font.set_variant(variant[k])
    t = text( 0.0, yp[k], variant[k], fontproperties=font,
              **alignment)

### Show weight options

weight = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']

for k in range(7):
    font = font0.copy()
    font.set_weight(weight[k])
    t = text( 0.4, yp[k], weight[k], fontproperties=font,
              **alignment)
### Show size options

```python
size  = ['xx-small', 'x-small', 'small', 'medium', 'large',
        'x-large', 'xx-large']

for k in range(7):
    font = font0.copy()
    font.set_size(size[k])
    t = text(0.8, yp[k], size[k], fontproperties=font,
             **alignment)
```

### Show bold italic

```python
font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
font.set_size('xx-small')
```

```python
t = text(0, 0.1, 'bold italic', fontproperties=font,
         **alignment)
```

```python
font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
font.set_size('medium')
```

```python
t = text(0, 0.2, 'bold italic', fontproperties=font,
         **alignment)
```

```python
font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
font.set_size('x-large')
```

```python
t = text(0, 0.3, 'bold italic', fontproperties=font,
         **alignment)
```

axis([-1,1,0,1])
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Same as fonts_demo using kwargs. If you prefer a more pythonic, OO style of coding, see examples/fonts_demo.py.

```python
#!/usr/bin/env python

from matplotlib.font_manager import FontProperties
from pylab import *

subplot(111, axisbg='w')

alignment = {'horizontalalignment':'center', 'verticalalignment':'baseline'}

### Show family options

family = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']

for k in range(5):
    if k == 2:
        t = text(-0.8, yp[k], family[k], family=family[k], family=family[k], size='large', **alignment)
```
name='Script MT', **alignment)
else:
t = text(-0.8, yp[k], family[k], family=family[k], **alignment)

### Show style options
style = ['normal', 'italic', 'oblique']
t = text(-0.4, 0.9, 'style', **alignment)
for k in range(3):
t = text(-0.4, yp[k], style[k], family='sans-serif', style=style[k], **alignment)

### Show variant options
variant= ['normal', 'small-caps']
t = text(0.0, 0.9, 'variant', **alignment)
for k in range(2):
t = text(0.0, yp[k], variant[k], family='serif', variant=variant[k], **alignment)

### Show weight options
weight = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']
t = text(0.4, 0.9, 'weight', **alignment)
for k in range(7):
t = text(0.4, yp[k], weight[k], weight=weight[k], **alignment)

### Show size options
size = ['xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large']
t = text(0.8, 0.9, 'size', **alignment)
for k in range(7):
t = text(0.8, yp[k], size[k], size=size[k], **alignment)

x = 0
### Show bold italic
t = text(x, 0.1, 'bold italic', style='italic', weight='bold', size='x-small', **alignment)
t = text(x, 0.2, 'bold italic', style='italic', weight='bold', size='medium', **alignment)
**alignment)

t = text(x, 0.3, 'bold italic',
        style='italic', weight='bold', size='x-large',
        **alignment)

axis([-1, 1, 0, 1])

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.90  *pylab_examples* example code: *ganged_plots.py*

```python
#!/usr/bin/env python

""
To create plots that share a common axes (visually) you can set the
hspace between the subplots close to zero (do not use zero itself).
Normally you'll want to turn off the tick labels on all but one of the
axes.

In this example the plots share a common *x*axis but you can follow the
```
same logic to supply a common y axis.

```python
from pylab import *

t = arange(0.0, 2.0, 0.01)
s1 = sin(2*pi*t)
s2 = exp(-t)
s3 = s1*s2

# axes rect in relative 0,1 coords left, bottom, width, height. Turn
# off xtick labels on all but the lower plot

f = figure()
subplots_adjust(hspace=0.001)

ax1 = subplot(311)
ax1.plot(t,s1)
yticks(arange(-0.9, 1.0, 0.4))
ylim(-1,1)

ax2 = subplot(312, sharex=ax1)
ax2.plot(t,s2)
yticks(arange(0.1, 1.0, 0.2))
ylim(0,1)

ax3 = subplot(313, sharex=ax1)
ax3.plot(t,s3)
yticks(arange(-0.9, 1.0, 0.4))
ylim(-1,1)

xticklabels = ax1.get_xticklabels()+ax2.get_xticklabels()
setp(xticklabels, visible=False)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
#np.seterr("raise")

from pylab import *

subplot(221, projection="aitoff")
title("Aitoff")
grid(True)

subplot(222, projection="hammer")
title("Hammer")
grid(True)

subplot(223, projection="lambert")
title("Lambert")
grid(True)

subplot(224, projection="mollweide")
title("Mollweide")
grid(True)

show()
84.92 pylab_examples example code: ginput_demo.py

[source code]

```python
# -*- noplot -*-

from __future__ import print_function

from pylab import arange, plot, sin, ginput, show

t = arange(10)
plot(t, sin(t))
print("Please click")
x = ginput(3)
print("clicked",x)
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.93 pylab_examples example code: ginput_manual_clabel.py

[source code]

```bash
#!/usr/bin/env python
# -*- noplot -*-

from __future__ import print_function

from __future__ import print_function

""
This provides examples of uses of interactive functions, such as ginput,
waitforbuttonpress and manual clabel placement.

This script must be run interactively using a backend that has a
graphical user interface (for example, using GTKAgg backend, but not
PS backend).

See also ginput_demo.py
""

import time
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

def tellme(s):
    print(s)
    plt.title(s, fontsize=16)
    plt.draw()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
# Define a triangle by clicking three points

plt.clf()
plt.axis([-1.,1.,-1.,1.])
plt.setp(plt.gca(),autoscale_on=False)

tellme('You will define a triangle, click to begin')

plt.waitforbuttonpress()

happy = False
while not happy:
    pts = []
    while len(pts) < 3:
        tellme('Select 3 corners with mouse')
        pts = np.asarray( plt.ginput(3,timeout=-1) )
        if len(pts) < 3:
            tellme('Too few points, starting over')
            time.sleep(1)  # Wait a second

    ph = plt.fill( pts[:,0], pts[:,1], 'r', lw=2 )

    tellme( 'Happy? Key click for yes, mouse click for no' )

    happy = plt.waitforbuttonpress()

    # Get rid of fill
    if not happy:
        for p in ph: p.remove()

# Now contour according to distance from triangle corners - just an example

# Define a nice function of distance from individual pts

def f(x,y,pts):
    z = np.zeros_like(x)
    for p in pts:
        z = z + 1/(np.sqrt((x-p[0])**2+(y-p[1])**2))
    return 1/z

X,Y = np.meshgrid( np.linspace(-1,1,51), np.linspace(-1,1,51) )
Z = f(X,Y,pts)

CS = plt.contour( X, Y, Z, 20 )

tellme('Use mouse to select contour label locations, middle button to finish')
CL = plt.clabel( CS, manual=True )

# Now do a zoom
Matplotlib, Release 1.4.3

```
# tellme( 'Now do a nested zoom, click to begin' )
plt.waitforbuttonpress()

happy = False
while not happy:
    tellme( 'Select two corners of zoom, middle mouse button to finish' )
    pts = np.asarray( plt.ginput(2,timeout=-1) )

    happy = len(pts) < 2
    if happy: break

    pts = np.sort(pts,axis=0)
    plt.axis( pts.T.ravel() )

tellme('All Done!')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.94 `pylab_examples` example code: `gradient_bar.py`
from matplotlib.pyplot import figure, show, cm
from numpy import arange
from numpy.random import rand

def gbar(ax, x, y, width=0.5, bottom=0):
    X = [[.6, .6], [.7, .7]]
    for left, top in zip(x, y):
        right = left + width
        ax.imshow(X, interpolation='bicubic', cmap=cm.Blues,
                  extent=(left, right, bottom, top), alpha=1)

fig = figure()
xmin, xmax = xlim = 0, 10
ymin, ymax = ylim = 0, 1
ax = fig.add_subplot(111, xlim=xlim, ylim=ylim,
                     autoscale_on=False)
X = [[.6, .6], [.7, .7]]
ax.imshow(X, interpolation='bicubic', cmap=cm.copper,
          extent=(xmin, xmax, ymin, ymax), alpha=1)

N = 10
x = arange(N) + 0.25
y = rand(N)
gbar(ax, x, y, width=0.7)
ax.set_aspect('auto')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
from numpy.random import uniform, seed
from matplotlib.mlab import griddata
import matplotlib.pyplot as plt
import numpy as np

# make up data.
#npts = int(input('enter # of random points to plot: '))
seed(0)
npts = 200
x = uniform(-2., 2., npts)
y = uniform(-2., 2., npts)
z = x*np.exp(-x**2 - y**2)

# define grid.
xi = np.linspace(-2.1, 2.1, 100)
yi = np.linspace(-2.1, 2.1, 200)

# grid the data.
zi = griddata(x, y, z, xi, yi, interp='linear')

# contour the gridded data, plotting dots at the nonuniform data points.
CS = plt.contour(xi, yi, zi, 15, linewidths=0.5, colors='k')
CS = plt.contourf(xi, yi, zi, 15, cmap=plt.cm.rainbow, vmax=abs(zi).max(), vmin=-abs(zi).max())

plt.colorbar()  # draw colorbar

# plot data points.
```

2134 Chapter 84. `pylab_examples` Examples
```python
plt.scatter(x, y, marker='o', c='b', s=5, zorder=10)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('griddata test (%d points) % npts')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**84.96 pylab_examples example code: hatch_demo.py**

```
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse, Polygon

fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.bar(range(1,5), range(1,5), color='red', edgecolor='black', hatch='/')
ax1.bar(range(1,5), [6] * 4, bottom=range(1,5), color='blue', edgecolor='black', hatch='//')
ax1.set_xticks([1.5, 2.5, 3.5, 4.5])
```

Hatching (pattern filled polygons) is supported currently in the PS, PDF, SVG and Agg backends only.

```
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse, Polygon

fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.bar(range(1,5), range(1,5), color='red', edgecolor='black', hatch='/')
ax1.bar(range(1,5), [6] * 4, bottom=range(1,5), color='blue', edgecolor='black', hatch='//')
ax1.set_xticks([1.5, 2.5, 3.5, 4.5])
```
ax2 = fig.add_subplot(132)
bars = ax2.bar(range(1,5), range(1,5), color='yellow', ecolor='black') + 
    ax2.bar(range(1, 5), [6] * 4, bottom=range(1,5), color='green', ecolor='black')
ax2.set_xticks([1.5,2.5,3.5,4.5])

patterns = ('-', '+', 'x', '\\', '*', 'o', '0', '.')
for bar, pattern in zip(bars, patterns):
    bar.set_hatch(pattern)

ax3 = fig.add_subplot(133)
ax3.fill([[1,3,3,1],[1,1,2,2]], fill=False, hatch='\\')
ax3.add_patch(Ellipse((4,1.5), 4, 0.5, fill=False, hatch='*'))
ax3.add_patch(Polygon([[0,0],[4,1.1],[6,2.5],[2,1.4]], closed=True, 
    fill=False, hatch='/'))
ax3.set_xlim((0,6))
ax3.set_ylim((0,2.5))

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.97 *pylab_examples* example code: *hexbin_demo.py*
hexbin is an axes method or pyplot function that is essentially a pcolor of a 2-D histogram with hexagonal cells. It can be much more informative than a scatter plot; in the first subplot below, try substituting 'scatter' for 'hexbin'.

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
n = 100000
x = np.random.standard_normal(n)
y = 2.0 + 3.0 * x + 4.0 * np.random.standard_normal(n)
xmin = x.min()
xmax = x.max()
ymin = y.min()
ymax = y.max()

plt.subplots_adjust(hspace=0.5)
plt.subplot(121)
plt.hexbin(x,y, cmap=plt.cm.YlOrRd_r)
plt.axis([xmin, xmax, ymin, ymax])
plt.title("Hexagon binning")
cb = plt.colorbar()
cb.set_label('counts')

plt.subplot(122)
plt.hexbin(x,y,bins='log', cmap=plt.cm.YlOrRd_r)
plt.axis([xmin, xmax, ymin, ymax])
plt.title("With a log color scale")
cb = plt.colorbar()
cb.set_label('log10(N)')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
hexbin is an axes method or pyplot function that is essentially a
color of a 2-D histogram with hexagonal cells.

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

delta = 0.025
x, y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians
x = X.ravel()
y = Y.ravel()
z = Z.ravel()

if 1:
    # make some points 20 times more common than others, but same mean
xcond = (-1 < x) & (x < 1)
ycond = (-2 < y) & (y < 0)
cond = xcond & ycond
xnew = x[cond]
ynew = y[cond]
znew = z[cond]
for i in range(20):
    x = np.hstack((x, xnew))
y = np.hstack((y, ynew))
z = np.hstack((z, znew))

xmin = x.min()
xmax = x.max()
ymin = y.min()
ymax = y.max()
gridsize = 30

plt.subplot(211)
plt.hexbin(x, y, C=z, gridsize=gridsize, marginals=True, cmap=plt.cm.RdBu,
          vmax=abs(z).max(), vmin=-abs(z).max())
plt.axis([xmin, xmax, ymin, ymax])
cb = plt.colorbar()
cb.set_label('mean value')

plt.subplot(212)
plt.hexbin(x, y, gridsize=gridsize, cmap=plt.cm.Blues_r)
plt.axis([xmin, xmax, ymin, ymax])
cb = plt.colorbar()
cb.set_label('N observations')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from pylab import *
x = randn(1000)
y = randn(1000)+5

#normal distribution center at x=0 and y=5
hist2d(x,y,bins=40)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from matplotlib.colors import LogNorm
from pylab import *

# normal distribution center at x=0 and y=5
x = randn(100000)
y = randn(100000)+5

hist2d(x, y, bins=40, norm=LogNorm())
colorbar()
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as colors

fig, ax = plt.subplots()
Ntotal = 1000
N, bins, patches = ax.hist(np.random.rand(Ntotal), 20)

# I'll color code by height, but you could use any scalar

# we need to normalize the data to 0..1 for the full range of the colormap
fracs = N.astype(float)/N.max()
norm = colors.Normalize(fracs.min(), fracs.max())

for thisfrac, thispatch in zip(fracs, patches):
    color = cm.jet(norm(thisfrac))
    thispatch.set_facecolor(color)
import matplotlib
from numpy.random import randn
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the default
tick locations.
    s = str(100 * y)
    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] == True:
        return s + r'$\%$
    else:
        return s + '%'

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.102  pylab_examples example code: histogram_percent_demo.py

import matplotlib
from numpy.random import randn
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the default
tick locations.
    s = str(100 * y)
    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] == True:
        return s + r'$\%$
    else:
        return s + '%'
x = randn(5000)

# Make a normed histogram. It'll be multiplied by 100 later.
plt.hist(x, bins=50, normed=True)

# Create the formatter using the function to_percent. This multiplies all the
# default labels by 100, making them all percentages
formatter = FuncFormatter(to_percent)

# Set the formatter
plt.gca().yaxis.set_major_formatter(formatter)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.103 pylab_examples example code: hyperlinks.py

[source code]

#!/usr/bin/env python
# -*- noplot -*-

""
This example demonstrates how to set a hyperlinks on various kinds of elements.

This currently only works with the SVG backend.
""

import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

f = plt.figure()
s = plt.scatter([1,2,3],[4,5,6])
s.set_urls(['http://www.bbc.co.uk/news','http://www.google.com','None'])
f.canvas.print_figure('scatter.svg')

f = plt.figure()
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2-Z1  # difference of Gaussians
im = plt.imshow(Z, interpolation='bilinear', cmap=cm.gray,
origin='lower', extent=[-3,3,-3,3])
```python
#!/usr/bin/env python
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
from matplotlib.path import Path
from matplotlib.patches import PathPatch

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2-Z1  # difference of Gaussians
```
path = Path([[0, 1], [1, 0], [0, -1], [-1, 0], [0, 1]])
patch = PathPatch(path, facecolor='none')
plt.gca().add_patch(patch)

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.gray,
                 origin='lower', extent=[-3, 3, -3, 3],
                 clip_path=patch, clip_on=True)
im.set_clip_path(patch)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.105 pylab_examples example code: image_demo.py

#!/usr/bin/env python
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2-Z1  # difference of Gaussians
im = plt.imshow(Z, interpolation='bicubic', cmap=cm.RdYlGn,
                 origin='lower', extent=[-3,3,-3,3],
                 vmax=abs(Z).max(), vmin=-abs(Z).max())

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.106  pylab_examples example code: image_demo2.py

#!/usr/bin/env python

from __future__ import print_function
from pylab import *
import matplotlib.cbook as cbook
w, h = 512, 512

datafile = cbook.get_sample_data('ct.raw.gz', asfileobj=True)
s = datafile.read()
A = fromstring(s, uint16).astype(float)
A *= 1.0/max(A)
A.shape = w, h

extent = (0, 25, 0, 25)
im = imshow(A, cmap=cm.hot, origin='upper', extent=extent)

markers = [(15.9, 14.5), (16.8, 15)]
x,y = zip(*markers)
plot(x, y, 'o')
#axis([0,25,0,25])

#axis('off')
title('CT density')

if 0:
x = asum(A,0)
subplot(212)
bar(arange(w), x)
xlim(0,h-1)
ylabel('density')
setp(gca(), 'xticklabels', [])

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.107  pylab_examples example code: image_interp.py
Chapter 84. `pylab_examples` Examples
#!/usr/bin/env python

The same (small) array, interpolated with three different interpolation methods.

The center of the pixel at A[i,j] is plotted at i+0.5, i+0.5. If you are using interpolation='nearest', the region bounded by (i,j) and (i+1,j+1) will have the same color. If you are using interpolation, the pixel center will have the same color as it does with nearest, but other pixels will be interpolated between the neighboring pixels.

Earlier versions of matplotlib (<0.63) tried to hide the edge effects from you by setting the view limits so that they would not be visible. A recent bugfix in antigrain, and a new implementation in the matplotlib._image module which takes advantage of this fix, no longer makes this necessary. To prevent edge effects, when doing interpolation, the matplotlib._image module now pads the input array with identical pixels around the edge. e.g., if you have a 5x5 array with colors a-y as below

```
a b c d e
f g h i j
k l m n o
p q r s t
```
the _image module creates the padded array,

```
  u v w x y
```

does the interpolation/resizing, and then extracts the central region. This allows you to plot the full range of your array w/o edge effects, and for example to layer multiple images of different sizes over one another with different interpolation methods - see examples/layer_images.py. It also implies a performance hit, as this new temporary, padded array must be created. Sophisticated interpolation also implies a performance hit, so if you need maximal performance or have very large images, interpolation='nearest' is suggested.

```python
from pylab import *
A = rand(5,5)
figure(1)
imshow(A, interpolation='nearest')
grid(True)

gc()
figure(2)
imshow(A, interpolation='bilinear')
grid(True)

gc()
figure(3)
imshow(A, interpolation='bicubic')
grid(True)

gc()
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
imshow with masked array input and out-of-range colors.

The second subplot illustrates the use of BoundaryNorm to get a filled contour effect.

```python
#!/usr/bin/env python

"""imshow with masked array input and out-of-range colors.

The second subplot illustrates the use of BoundaryNorm to get a filled contour effect.
""

from pylab import *
from numpy import ma
import matplotlib.colors as colors

delta = 0.025
x = y = arange(-3.0, 3.0, delta)
X, Y = meshgrid(x, y)
Z1 = bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10 * (Z2-Z1)  # difference of Gaussians

# Set up a colormap:
palette = cm.gray
palette.set_over('r', 1.0)
palette.set_under('g', 1.0)
```
palette.set_bad('b', 1.0)
# Alternatively, we could use
# palette.set_bad(alpha = 0.0)
# to make the bad region transparent. This is the default.
# If you comment out all the palette.set* lines, you will see
# all the defaults; under and over will be colored with the
# first and last colors in the palette, respectively.
Zm = ma.masked_where(Z > 1.2, Z)

# By setting vmin and vmax in the norm, we establish the
# range to which the regular palette color scale is applied.
# Anything above that range is colored based on palette.set_over, etc.

subplot(1,2,1)
im = imshow(Zm, interpolation='bilinear',
            cmap=palette,
            norm = colors.Normalize(vmin = -1.0, vmax = 1.0, clip = False),
            origin='lower', extent=[-3,3,-3,3])
title('Green=low, Red=high, Blue=bad')
colorbar(im, extend='both', orientation='horizontal', shrink=0.8)

subplot(1,2,2)
im = imshow(Zm, interpolation='nearest',
            cmap=palette,
            norm = colors.BoundaryNorm([-1, -0.5, -0.2, 0, 0.2, 0.5, 1],
                                        ncolors=256, clip = False),
            origin='lower', extent=[-3,3,-3,3])
title('With BoundaryNorm')
colorbar(im, extend='both', spacing='proportional',
            orientation='horizontal', shrink=0.8)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This illustrates the NonUniformImage class, which still needs an axes method interface; either a separate interface, or a generalization of imshow.

```python
from matplotlib.pyplot import figure, show
import numpy as np
from matplotlib.image import NonUniformImage
from matplotlib import cm

interp='nearest'

x = np.linspace(-4, 4, 9)
x2 = x**3
y = np.linspace(-4, 4, 9)
# print 'Size %d points' % (len(x) * len(y))
z = np.sqrt(x[np.newaxis, :]*x2 + y[:,np.newaxis]*y2)

fig = figure()
fig.suptitle('NonUniformImage class')
ax = fig.add_subplot(221)
```

```
NonUniformImage class nearest

```

```
NonUniformImage class bilinear

```

```
NonUniformImage class nearest

```

```
NonUniformImage class bilinear

```
im = NonUniformImage(ax, interpolation=interp, extent=(-4,4,-4,4),
cmap=cm.Purples)
im.set_data(x, y, z)
ax.images.append(im)
ax.set_xlim(-4,4)
ax.set_ylim(-4,4)
ax.set_title(interp)

ax = fig.add_subplot(222)
im = NonUniformImage(ax, interpolation=interp, extent=(-64,64,-4,4),
cmap=cm.Purples)
im.set_data(x2, y, z)
ax.images.append(im)
ax.set_xlim(-64,64)
ax.set_ylim(-4,4)
ax.set_title(interp)

interp = 'bilinear'

ax = fig.add_subplot(223)
im = NonUniformImage(ax, interpolation=interp, extent=(-4,4,-4,4),
cmap=cm.Purples)
im.set_data(x, y, z)
ax.images.append(im)
ax.set_xlim(-4,4)
ax.set_ylim(-4,4)
ax.set_title(interp)

ax = fig.add_subplot(224)
im = NonUniformImage(ax, interpolation=interp, extent=(-64,64,-4,4),
cmap=cm.Purples)
im.set_data(x2, y, z)
ax.images.append(im)
ax.set_xlim(-64,64)
ax.set_ylim(-4,4)
ax.set_title(interp)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
You can specify whether images should be plotted with the array origin
x[0,0] in the upper left or upper right by using the origin parameter.
You can also control the default be setting image.origin in your
matplotlibrc file; see http://matplotlib.org/matplotlibrc
"""

from pylab import *

x = arange(100.0); x.shape = 10,10

interp = 'bilinear';
#interp = 'nearest';
lim = -2,11,-2,6

subplot(211, axisbg='g')
title('blue should be up')
imshow(x, origin='upper', interpolation=interp)
#axis(lim)

subplot(212, axisbg='y')
title('blue should be down')
imshow(x, origin='lower', interpolation=interp)
from __future__ import print_function
import numpy
from matplotlib.pyplot import figure, show

class IndexTracker:
    def __init__(self, ax, X):
        self.ax = ax
        ax.set_title('use scroll wheel to navigate images')

        self.X = X
        rows, cols, self.slices = X.shape
        self.ind = self.slices/2
```python
self.im = ax.imshow(self.X[:,:,self.ind])
self.update()

def onscroll(self, event):
    print ("%s %s" % (event.button, event.step))
    if event.button== 'up':
        self.ind = numpy.clip(self.ind+1, 0, self.slices-1)
    else:
        self.ind = numpy.clip(self.ind-1, 0, self.slices-1)
    self.update()

def update(self):
    self.im.set_data(self.X[:,:,self.ind])
    ax.set_ylabel('slice %s' %self.ind)
    self.im.axes.figure.canvas.draw()

fig = figure()
ax = fig.add_subplot(111)

X = numpy.random.rand(20,20,40)
tracker = IndexTracker(ax, X)

fig.canvas.mpl_connect('scroll_event', tracker.onscroll)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)```
import matplotlib.pyplot as plt
from numpy import pi, sin, linspace
from matplotlib.mlab import stineman_interp

x = linspace(0,2*pi,20);
y = sin(x); yp = None
xi = linspace(x[0],x[-1],100);
yi = stineman_interp(xi,x,y,yp);

fig, ax = plt.subplots()
ax.plot(x,y,'ro',xi,yi,'-b.')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
You can use decreasing axes by flipping the normal order of the axis limits
"""

from pylab import *

t = arange(0.01, 5.0, 0.01)
s = exp(-t)
plot(t, s)
xlim(5, 0)   # decreasing time
xlabel('decreasing time (s)')
ylabel('voltage (mV)')
title('Should be growing...')
grid(True)
show()
Layer images above one another using alpha blending

```python
#!/usr/bin/env python
#
# Layer images above one another using alpha blending
#
from __future__ import division
from pylab import *

def func3(x, y):
    return (1 - x/2 + x**5 + y**3)*exp(-x**2-y**2)

# make these smaller to increase the resolution
dx, dy = 0.05, 0.05
x = arange(-3.0, 3.0, dx)
y = arange(-3.0, 3.0, dy)
X, Y = meshgrid(x, y)

# when layering multiple images, the images need to have the same
# extent. This does not mean they need to have the same shape, but
```
# they both need to render to the same coordinate system determined by
# xmin, xmax, ymin, ymax. Note if you use different interpolations
# for the images their apparent extent could be different due to
# interpolation edge effects

xmin, xmax, ymin, ymax = amin(x), amax(x), amin(y), amax(y)
extent = xmin, xmax, ymin, ymax
fig = plt.figure(frameon=False)

Z1 = array(([0,1]*4 + [1,0]*4)*4); Z1.shape = 8,8  # chessboard
im1 = imshow(Z1, cmap=cm.gray, interpolation='nearest',
              extent=extent)
hold(True)

Z2 = func3(X, Y)

im2 = imshow(Z2, cmap=cm.jet, alpha=.9, interpolation='bilinear',
              extent=extent)
#axis([xmin, xmax, ymin, ymax])

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
# Make a legend for specific lines.
import matplotlib.pyplot as plt
import numpy as np

t1 = np.arange(0.0, 2.0, 0.1)
t2 = np.arange(0.0, 2.0, 0.01)

# note that plot returns a list of lines. The "ll, = plot" usage
# extracts the first element of the list into ll using tuple
# unpacking. So ll is a Line2D instance, not a sequence of lines
l1, = plt.plot(t2, np.exp(-t2))
l2, l3 = plt.plot(t2, np.sin(2 * np.pi * t2), '--go', t1, np.log(1 + t1), '.')
l4, = plt.plot(t2, np.exp(-t2) * np.sin(2 * np.pi * t2), 'rs-.' )

plt.legend( (l2, l4), ('oscillatory', 'damped'), loc='upper right', shadow=True)
plt.xlabel('time')
plt.ylabel('volts')
plt.title('Damped oscillation')
plt.show()
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 1)

# Plot the lines y=x^n for n=1..4.
ax = plt.subplot(2, 1, 1)
for n in range(1, 5):
    plt.plot(x, x**n, label="n={}".format(n))
plt.legend(loc="upper left", bbox_to_anchor=[0, 1],
          ncol=2, shadow=True, title="Legend", fancybox=True)
ax.get_legend().get_title().set_color("red")

# Demonstrate some more complex labels.
ax = plt.subplot(2, 1, 2)
plt.plot(x, x**2, label="multi
line")
half_pi = np.linspace(0, np.pi / 2)
plt.plot(np.sin(half_pi), np.cos(half_pi), label=r"$\frac{1}{2}\pi$")
plt.plot(x, 2**(x**2), label="$2^{x^2}$")
plt.legend(shadow=True, fancybox=True)

plt.show()
import matplotlib.pyplot as plt

fig, axes = plt.subplots(3, 1)
top_ax, middle_ax, bottom_ax = axes

top_ax.bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4, label="Bar 1", align="center")
top_ax.bar([0.5, 1.5, 2.5], [0.3, 0.2, 0.2], color="red", width=0.4, label="Bar 2", align="center")
top_ax.legend()

middle_ax.errorbar([0, 1, 2], [2, 3, 1], xerr=0.4, fmt="s", label="test 1")
middle_ax.errorbar([0, 1, 2], [3, 2, 4], yerr=0.3, fmt="o", label="test 2")
middle_ax.errorbar([0, 1, 2], [1, 1, 3], xerr=0.4, yerr=0.3, fmt="^", label="test 3")
middle_ax.legend()

bottom_ax.stem([0.3, 1.5, 2.7], [1, 3.6, 2.7], label="stem test")
bottom_ax.legend()
```python
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.colors import colorConverter

import numpy as np

# In order to efficiently plot many lines in a single set of axes,
# Matplotlib has the ability to add the lines all at once. Here is a
# simple example showing how it is done.

x = np.arange(100)
# Here are many sets of y to plot vs x
ys = x[:50, np.newaxis] + x[np.newaxis, :]

segs = np.zeros((50, 100, 2), float)
segs[:,:,1] = ys
segs[:,:,0] = segs[:,:,1]
```

---

84.118. `pylab_examples` example code: `line_collection.py`
segs[:,:,0] = x

# Mask some values to test masked array support:
segs = np.ma.masked_where((segs > 50) & (segs < 60), segs)

# We need to set the plot limits.
ax = plt.axes()
ax.set_xlim(x.min(), x.max())
ax.set_ylim(ys.min(), ys.max())

# colors is sequence of rgba tuples
# linestyle is a string or dash tuple. Legal string values are
# solid|dashed|dashdot|dotted. The dash tuple is (offset, onoffseq)
# where onoffseq is an even length tuple of on and off ink in points.
# If linestyle is omitted, 'solid' is used
# See matplotlib.collections.LineCollection for more information
line_segments = LineCollection(segs,
    linewidths = (0.5,1,1.5,2),
    colors = [colorConverter.to_rgba(i) \
              for i in ('b','g','r','c','m','y','k')],
    linestyle = 'solid')

ax.add_collection(line_segments)
ax.set_title('Line collection with masked arrays')
plt.show()
from matplotlib.collections import LineCollection

# In order to efficiently plot many lines in a single set of axes, 
# Matplotlib has the ability to add the lines all at once. Here is a 
# simple example showing how it is done.

N = 50
x = arange(N)
# Here are many sets of y to plot vs x
ys = [x+i for i in x]

# We need to set the plot limits, they will not autoscale
ax = axes()
ax.set_xlim((amin(x),amax(x)))
ax.set_ylim((amin(amin(ys)),amax(amax(ys))))

# colors is sequence of rgba tuples
# linestyle is a string or dash tuple. Legal string values are
# solid|dashed|dashdot|dotted. The dash tuple is (offset, onoffseq)
# where onoffseq is an even length tuple of on and off ink in points.
# If linestyle is omitted, 'solid' is used
# See matplotlib.collections.LineCollection for more information
line_segments = LineCollection([[list(zip(x,y)) for y in ys], # Make a sequence of x,y pairs
                              linewidths  = (0.5,1.5,2),
                              linestyles = 'solid')

line_segments.set_array(x)
ax.add_collection(line_segments)
fig = gcf()
axcb = fig.colorbar(line_segments)
axcb.set_label('Line Number')
ax.set_title('Line Collection with mapped colors')
scl(line_segments) # This allows interactive changing of the colormap.
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.120  pylab_examples example code: load_converter.py
```python
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading', datafile)

dates, closes = np.loadtxt(
    datafile, delimiter=',',
    converters={0: strpdate2num('%d-%b-%y')},
    skiprows=1, usecols=(0, 2), unpack=True)

fig = figure()
ax = fig.add_subplot(111)
ax.plot_date(dates, closes, '-')
fig.autofmt_xdate()
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```

84.121  pylab_examples example code: loadrec.py

![Graph showing stock prices over time.](image-url)

```python
from __future__ import print_function
from matplotlib import mlab
```

84.121.  pylab_examples example code: loadrec.py 2171
from pylab import figure, show
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading', datafile)
a = mlab.csv2rec(datafile)
a.sort()
print(a.dtype)

fig = figure()
ax = fig.add_subplot(111)
ax.plot(a.date, a.adj_close, '-

fig.autofmt_xdate()

# if you have xlwt installed, you can output excel
try:
    import mpl_toolkits.exceltools as exceltools
    exceltools.rec2excel(a, 'test.xls')
except ImportError:
    pass

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!usr/bin/env python

from matplotlib import
pylab

data = ((3,1000), (10,3), (100,30), (500, 800), (50,1))

pylab.xlabel("FOO")
pylab.ylabel("FOO")
pylab.title("Testing")
pylab.gca().set_yscale('log')

dim = len(data[0])
w = 0.75
dimw = w / dim

x = pylab.arange(len(data))
for i in range(len(data[0])):
y = [d[i] for d in data]
b = pylab.bar(x + i * dimw, y, dimw, bottom=0.001)
pylab.gca().set_xticks(x + w / 2)
pylab.gca().set_ylim((0.001,1000))

# pylab_examples example code: log_bar.py
pylab.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.123 pylab_examples example code: log_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt

plt.subplots_adjust(hspace=0.4)
t = np.arange(0.01, 20.0, 0.01)

# log y axis
plt.subplot(221)
plt.semilogy(t, np.exp(-t/5.0))
plt.title('semilogy')
plt.grid(True)

# log x axis
plt.subplot(222)
plt.semilogx(t, np.sin(2*np.pi*t))
plt.title('semilogx')

# loglog base 4 on x
plt.subplot(223)
plt.loglog(t, np.exp(-t/5.0))
plt.title('loglog base 4 on x')

# Errorbars go negative
plt.subplot(224)
plt.errorbar(t, np.sin(2*np.pi*t), np.sin(2*np.pi*0.5*t))
plt.title('Errorbars go negative')
```

Chapter 84. pylab_examples Examples
plt.grid(True)

# log x and y axis
plt.subplot(223)
plt.loglog(t, 20*np.exp(-t/10.0), basex=2)
plt.grid(True)
plt.title('loglog base 4 on x')

# with errorbars: clip non-positive values
ax = plt.subplot(224)
ax.set_xscale("log", nonposx='clip')
ax.set_yscale("log", nonposy='clip')

x = 10.0**np.linspace(0.0, 2.0, 20)
y = x**2.0
plt.errorbar(x, y, xerr=0.1*x, yerr=5.0+0.75*y)
ax.set_ylim(ymin=0.1)
ax.set_title('Errorbars go negative')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
from pylab import *

dt = 0.01
t = arange(dt, 20.0, dt)

semilogx(t, exp(-t/5.0))
grid(True)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
# This file generates the matplotlib web page logo

from __future__ import print_function
from pylab import *
import matplotlib.cbook as cbook

# convert data to mV
datafile = cbook.get_sample_data('membrane.dat', asfileobj=False)
print('loading', datafile)

x = 1000*0.1*fromstring(file(datafile, 'rb').read(), float32)
# 0.0005 is the sample interval
t = 0.0005*arange(len(x))
figure(1, figsize=(7,1), dpi=100)
ax = subplot(111, axisbg='y')
plot(t, x)
text(0.5, 0.5, 'matplotlib', color='r', fontsize=40, fontname='Courier',
     horizontalalignment='center',
     verticalalignment='center',
     transform=ax.transAxes,
)
axis([1, 1.72,-60, 10])
setp(gca(), 'xticklabels', [])
setp(gca(), 'yticklabels', [])

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
Demonstrate how to use major and minor tickers.

The two relevant userland classes are Locators and Formatters. Locators determine where the ticks are and formatters control the formatting of ticks.

Minor ticks are off by default (NullLocator and NullFormatter). You can turn minor ticks on w/o labels by setting the minor locator. You can also turn labeling on for the minor ticker by setting the minor formatter.

Make a plot with major ticks that are multiples of 20 and minor ticks that are multiples of 5. Label major ticks with %d formatting but don't label minor ticks.

The MultipleLocator ticker class is used to place ticks on multiples of some base. The FormatStrFormatter uses a string format string (e.g., %d, %1.2f or %1.1f cm) to format the tick.

The pylab interface grid command changes the grid settings of the
major ticks of the y and y axis together. If you want to control the grid of the minor ticks for a given axis, use for example

    ax.xaxis.grid(True, which='minor')

Note, you should not use the same locator between different Axis because the locator stores references to the Axis data and view limits

""

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import MultipleLocator, FormatStrFormatter

majorLocator = MultipleLocator(20)
majorFormatter = FormatStrFormatter('%d')
minorLocator = MultipleLocator(5)

t = np.arange(0.0, 100.0, 0.1)
s = np.sin(0.1*np.pi*t)*np.exp(-t*0.01)

fig, ax = plt.subplots()
plt.plot(t,s)
ax.xaxis.set_major_locator(majorLocator)
ax.xaxis.set_major_formatter(majorFormatter)

# for the minor ticks, use no labels; default NullFormatter
ax.xaxis.set_minor_locator(minorLocator)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
Automatic tick selection for major and minor ticks.

Use interactive pan and zoom to see how the tick intervals change. There will be either 4 or 5 minor tick intervals per major interval, depending on the major interval.
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import AutoMinorLocator

# One can supply an argument to AutoMinorLocator to
# specify a fixed number of minor intervals per major interval, e.g.:
# minorLocator = AutoMinorLocator(2)
# would lead to a single minor tick between major ticks.

minorLocator = AutoMinorLocator()

t = np.arange(0.0, 100.0, 0.01)
s = np.sin(2*np.pi*t)*np.exp(-t*0.01)

fig, ax = plt.subplots()
plt.plot(t, s)
ax.xaxis.set_minor_locator(minorLocator)

plt.tick_params(which='both', width=2)
plt.tick_params(which='major', length=7)
plt.tick_params(which='minor', length=4, color='r')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

The techniques here are no longer required with the new support for spines in matplotlib -- see http://matplotlib.org/examples/pylab_examples/spine_placement_demo.html.
This example should be considered deprecated and is left just for demo purposes for folks wanting to make a pseudo-axis

```python
import numpy as np
from pylab import figure, show
import matplotlib.lines as lines

def make_xaxis(ax, yloc, offset=0.05, **props):
    xmin, xmax = ax.get_xlim()
    locs = [loc for loc in ax.xaxis.get_majorticklocs()
            if loc>=xmin and loc<=xmax]
    tickline, = ax.plot(locs, [yloc]*len(locs), linestyle='',
                        marker=lines.TICKDOWN, **props)
    axline, = ax.plot([xmin, xmax], [yloc, yloc], **props)
    tickline.set_clip_on(False)
    axline.set_clip_on(False)
    for loc in locs:
        ax.text(loc, yloc-offset, '

        def make_yaxis(ax, xloc=0, offset=0.05, **props):
            ymin, ymax = ax.get_ylim()
            locs = [loc for loc in ax.yaxis.get_majorticklocs()
                    if loc>=ymin and loc<=ymax]
            tickline, = ax.plot([xloc]*len(locs), locs, linestyle='',
                                marker=lines.TICKLEFT, **props)
            axline, = ax.plot([xloc, xloc], [ymin, ymax], **props)
            tickline.set_clip_on(False)
            axline.set_clip_on(False)
    
        for loc in locs:
            ax.text(xloc-offset, loc, '

        props = dict(color='black', linewidth=2, markeredgewidth=2)
        x = np.arange(200.)
        y = np.sin(2*np.pi*x/200.) + np.random.rand(200)-0.5
        fig = figure(facecolor='white')
        ax = fig.add_subplot(111, frame_on=False)
        ax.axison = False
        ax.plot(x, y, 'd', markersize=8, markerfacecolor='blue')
        ax.set_xlim(0, 200)
        ax.set_ylim(-1.5, 1.5)
        make_xaxis(ax, 0, offset=0.1, **props)
        make_yaxis(ax, 0, offset=5, **props)
        show()
```
import matplotlib.pyplot as plt
import matplotlib.path as mpath
import numpy as np

star = mpath.Path.unit_regular_star(6)
circle = mpath.Path.unit_circle()
# concatenate the circle with an internal cutout of the star
verts = np.concatenate([circle.vertices, star.vertices[::-1, ...]])
codes = np.concatenate([circle.codes, star.codes])
cut_star = mpath.Path(verts, codes)

plt.plot(np.arange(10)**2, '--r', marker=cut_star, markersize=15)
plt.show()
84.130  pylab_examples example code: markevery_demo.py
Examples of different mark every parameters:

- `markevery=None`
- `markevery=8`
- `markevery=(30, 8)`
- `markevery=[16, 24, 30]`
- `markevery=[0, -1]`
- `markevery=slice(100, 200, 3)`
- `markevery=0.1`
- `markevery=0.3`
- `markevery=1.5`
- `markevery=(0.0, 0.1)`
- `markevery=(0.45, 0.1)`

The plots illustrate how different mark every parameters affect the visualization of data points on a log-log scale.
This example demonstrates the various options for showing a marker at a subset of data points using the `markevery` property of a Line2D object.

Integer arguments are fairly intuitive. e.g. `markevery`=5 will plot every 5th marker starting from the first data point.

Float arguments allow markers to be spaced at approximately equal distances along the line. The theoretical distance along the line between markers is determined by multiplying the display-coordinate distance of the axes bounding-box diagonal by the value of `markevery`. The data points closest to the theoretical distances will be shown.

A slice or list/array can also be used with `markevery` to specify the markers to show.

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
```
import matplotlib.gridspec as gridspec

#define a list of markevery cases to plot
cases = [None,
         8,
         (30, 8),
         [16, 24, 30], [0,-1],
         slice(100,200,3),
         0.1, 0.3, 1.5,
         (0.0, 0.1), (0.45, 0.1)]

#define the figure size and grid layout properties
figsize = (10, 8)
cols = 3
gs = gridspec.GridSpec(len(cases) // cols + 1, cols)

#define the data for cartesian plots
delta = 0.11
x = np.linspace(0, 10 - 2 * delta, 200) + delta
y = np.sin(x) + 1.0 + delta

#plot each markevery case for linear x and y scales
fig1 = plt.figure(num=1, figsize=figsize)
ax = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    ax.append(fig1.add_subplot(gs[row, col]))
    ax[-1].set_title('markevery=%s' % str(case))
    ax[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)

#fig1.tight_layout()

#plot each markevery case for log x and y scales
fig2 = plt.figure(num=2, figsize=figsize)
axlog = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    axlog.append(fig2.add_subplot(gs[row, col]))
    axlog[-1].set_xscale('log')
    axlog[-1].set_yscale('log')
    axlog[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)

fig2.tight_layout()

#plot each markevery case for linear x and y scales but zoomed in
#note the behaviour when zoomed in. When a start marker offset is specified
#it is always interpreted with respect to the first data point which might be
#different to the first visible data point.
fig3 = plt.figure(num=3, figsize=figsize)
axzoom = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    axzoom.append(fig3.add_subplot(gs[row, col]))
    axzoom[-1].set_title('markevery=%s' % str(case))
    axzoom[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)

fig3.tight_layout()
```
col = i % cols
axzoom.append(fig3.add_subplot(gs[row, col]))
axzoom[-1].set_title('markevery=%s' % str(case))
axzoom[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
axzoom[-1].set_xlim((6, 6.7))
axzoom[-1].set_ylim((1.1, 1.7))
fig3.tight_layout()

# define data for polar plots
r = np.linspace(0, 3.0, 200)
theta = 2 * np.pi * r

# plot each markevery case for polar plots
fig4 = plt.figure(num=4, figsize=figsize)
axpolar = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    axpolar.append(fig4.add_subplot(gs[row, col], polar=True))
    axpolar[-1].set_title('markevery=%s' % str(case))
    axpolar[-1].plot(theta, r, 'o', ls='-', ms=4, markevery=case)
fig4.tight_layout()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

'''
Plot lines with points masked out.
This would typically be used with gappy data, to break the line at the data gaps.
'''

from pylab import *

x = ma.arange(0, 2*pi, 0.02)
y = ma.sin(x)
y1 = sin(2*x)
y2 = sin(3*x)
ym1 = ma.masked_where(y1 > 0.5, y1)
ym2 = ma.masked_where(y2 < -0.5, y2)

lines = plot(x, y, 'r', x, ym1, 'g', x, ym2, 'bo')
setp(lines[0], linewidth = 4)
setp(lines[1], linewidth = 2)
setp(lines[2], markersize = 10)
legend( ('No mask', 'Masked if > 0.5', 'Masked if < -0.5'),
        loc = 'upper right')
title('Masked line demo')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.132 pylab_examples example code: mathtext_demo.py

#!/usr/bin/env python

""
Use matplotlib's internal LaTeX parser and layout engine. For true
latex rendering, see the text.usetex option
""

import numpy as np
from matplotlib.pyplot import figure, show

fig = figure()
fig.subplots_adjust(bottom=0.2)

ax = fig.add_subplot(111, axisbg='y')
ax.plot([1,2,3], 'r')
ax.plot([1,2,3], 'r')
x = np.arange(0.0, 3.0, 0.1)

ax.grid(True)
ax.set_xlabel(r'$\Delta_i^j$', fontsize=20)
ax.set_ylabel(r'$\Delta_{i+1}^j$', fontsize=20)
tex = r'$\mathcal{R}\prod_{i=\alpha_{i+1}}^{\infty} a_i\sin(2 \pi f x_i)$'

ax.text(1, 1.6, tex, fontsize=20, va='bottom')

ax.legend([r'"$\sqrt{x^2}$"'])

ax.set_title(r'$\Delta_i^j \hspace{0.4} \mathrm{versus} \hspace{0.4} \Delta_{i+1}^j$', fontsize=20)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Matplotlib's math rendering engine

\[
W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8\pi^2} \int_{\alpha_2}^{\alpha_2'} d\alpha_2' \left[ \frac{U^{2\beta}_{\delta_1 \rho_1} - \alpha_2' U^{1\beta}_{\rho_1 \sigma_2}}{U^{0\beta}_{\rho_1 \sigma_2}} \right]
\]

Subscripts and superscripts:
\[\alpha_i > \beta_i, \alpha^j_i + 1 = \sin(2\pi f_j t_i) e^{-5t_i/\tau}, \ldots\]

Fractions, binomials and stacked numbers:
\[
\frac{3}{4}, \left( \frac{3}{4} \right)^3, \frac{3}{4}, \left( \frac{5}{4} - \frac{1}{4} \right), \ldots
\]

Radicals:
\[
\sqrt{2}, \sqrt[3]{x}, \ldots
\]

Fonts:
Roman, Italic, Typewriter or CALLIGRAPHY

Accents:
\[\acute{a}, \bar{a}, \breve{a}, \grave{a}, \check{a}, \ddot{a}, \hat{a}, \tilde{a}, \vec{a}, \hat{xyz}, \tilde{xyz}, \ldots\]

Greek, Hebrew:
\[\alpha, \beta, \chi, \delta, \lambda, \mu, \Delta, \Gamma, \Omega, \Phi, \Pi, \Upsilon, \nabla, \Re, \Im, \ldots\]

Delimiter, functions and Symbols:
\[
\Pi, \int, \oint, \sum, \log, \sin, \sim, \oplus, \star, \alpha, \infty, \partial, \Re, \leftrightarrow, \ldots
\]

""
Selected features of Matplotlib's math rendering engine.
"""

from __future__ import print_function
import matplotlib.pyplot as plt
import os
import sys
import re
import gc

# Selection of features following "Writing mathematical expressions" tutorial
mathtext_titles = {
    0: "Header demo",
    1: "Subscripts and superscripts",
    2: "Fractions, binomials and stacked numbers",
    3: "Radicals",
    4: "Fonts",
    5: "Accents",
    6: "Greek, Hebrew",
    7: "Delimiters, functions and Symbols"}

n_lines = len(mathtext_titles)

# Randomly picked examples
mathtext_demos = {
    0: r'$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = ' +
      r'U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2} ' +
      r'\int^\alpha_2_{\alpha_2} d \alpha^{\prime}_2 \left[ ' +
      r'U^{2\beta}_{\delta_1 \rho_1} - \alpha^{\prime}_2U^{1\beta}_{\rho_1 ' +
      r'\sigma_2}\right] ' +
      r'/U^{0\beta}_{\rho_1 \sigma_2}$",
    1: r'$\alpha_i > \beta_i, ' +
      r'\alpha_{i+1}^j = {\rm sin}(2\pi f_j t_i) e^{-5 t_i/\tau}, ' +
      r'\ldots$",
    2: r'$\frac{3}{4}, \binom{3}{4}, \stackrel{3}{4}, ' +
      r'\left(\frac{5 - \frac{1}{x}}{4}\right), \ldots$",
    3: r'$\sqrt{2}, \sqrt[3]{x}, \ldots$",
    4: r'$\mathrm{Roman}, \mathit{Italic}, ' +
      r'\mathtt{Typewriter} \ or \ \mathcal{CALLIGRAPHY}$",
    5: r'$\acute a, \bar a, \breve a, \dot a, \ddot a, ' +
      r'\grave a, \hat a, \tilde a, \vec a, \widehat{xyz}, ' +
      r'\widetilde{xyz}, \ldots$",
    6: r'$\alpha, \beta, \chi, \delta, ' +
      r'\lambda, \mu, ' +
      r'\Delta, \Gamma, \Omega, \Phi, \Pi, ' +
      r'\Upsilon, \nabla, ' +
      r'\aleph, \beth, \daleth, \gimel, ' +
      r'\coprod, \int, \oint, \prod, ' +
      r'\sum, ' +
      r'\log, \sin, \approx, \oplus, ' +
      r'\varpropto, ' +
      r'\infty, \partial, \Re, ' +
      r'\leftrightarrow, \ldots$"
}

def doall():
    # Colors used in mpl online documentation.
    mpl_blue_rgb = (191./255., 209./256., 212./255.)
    mpl_orange_rgb = (202/255., 121/256., 0./255.)
    mpl_grey_rgb = (51./255., 51./255., 51./255.)
# Creating figure and axis.
plt.figure(figsize=(6, 7))
plt.axes([0.01, 0.01, 0.98, 0.90], axisbg="white", frameon=True)
plt.gca().set_xlim(0., 1.)
plt.gca().set_ylim(0., 1.)
plt.gca().set_title("Matplotlib's math rendering engine",
                    color=mpl_grey_rvb, fontsize=14, weight='bold')
plt.gca().set_xticklabels("", visible=False)
plt.gca().set_yticklabels("", visible=False)

# Gap between lines in axes coords
line_axesfrac = (1. / (n_lines))

# Plotting header demonstration formula
full_demo = mathext_demos[0]
plt.annotate(full_demo,
             xy=(0.5, 1. - 0.59*line_axesfrac),
             xycoords='data', color=mpl_orange_rvb, ha='center',
             fontsize=20)

# Plotting features demonstration formulae
for i_line in range(1, n_lines):
    baseline = 1. - (i_line)*line_axesfrac
    baseline_next = baseline - line_axesfrac*1.
    title = mathtext_titles[i_line] + ":
    fill_color = ['white', mpl_blue_rvb[i_line % 2]]
    plt.fill_between([0., 1.], [baseline, baseline],
                     [baseline_next, baseline_next],
                     color=fill_color, alpha=0.5)
    plt.annotate(title,
                 xy=(0.07, baseline - 0.3*line_axesfrac),
                 xycoords='data', color=mpl_grey_rvb, ha='center',
                 fontsize=16)
    demo = mathext_demos[i_line]
    plt.annotate(demo,
                 xy=(0.05, baseline - 0.75*line_axesfrac),
                 xycoords='data', color=mpl_grey_rvb,
                 fontsize=16)

for i in range(n_lines):
    s = mathext_demos[i]
    print(i, s)
plt.show()

if '--latex' in sys.argv:
    # Run: python mathtext_examples.py --latex
    # Need amsmath and amssymb packages.
    fd = open("mathtext_examples.ltx", "w")
    fd.write("\documentclass{article}\n")
    fd.write("\usepackage{amsmath, amssymb}\n")
    fd.write("\begin{document}\n")
    fd.write("\begin{enumerate}\n")
    for i in range(n_lines):
        s = mathext_demos[i]
        print(i, s)
s = mathext_demos[i]
s = re.sub(r"(?<!\$)\\$, "$", s)
fd.write("\item %s
" % s)

fd.write("\end{enumerate}\n")
fd.write("\end{document}\n")
fd.close()

else:
doall()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.134 pylab_examples example code: matplotlib_icon.py

#!/usr/bin/env python

""
make the matplotlib svg minimization icon
""

import matplotlib
#matplotlib.use('Svg')
from pylab import *

rc('grid', ls='-', lw=2, color='k')
fig = figure(figsize=(1, 1), dpi=72)
axes([0.025, 0.025, 0.95, 0.95], axisbg='#bfd1d4')

t = arange(0, 2, 0.05)
s = sin(2*pi*t)
plot(t, s, linewidth=4, color='#ca7900')
axis([-0.2, 2.2, -1.2, 1.2])

# grid(True)
setp(gca(), xticklabels=[], yticklabels=[]) 
savefig('matplotlib', facecolor='0.75')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.135  pylab_examples example code: matshow.py
"""Simple matshow() example."""

```python
from matplotlib.pylab import *

def samplemat(dims):
    """Make a matrix with all zeros and increasing elements on the diagonal"""
    aa = zeros(dims)
    for i in range(min(dims)):
        aa[i, i] = i
    return aa

# Display 2 matrices of different sizes
dimlist = [(12, 12), (15, 35)]
for d in dimlist:
    matshow(samplemat(d))

# Display a random matrix with a specified figure number and a grayscale colormap
matshow(rand(64, 64), fignum=100, cmap=cm.gray)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
# -*- noplot -*-

from __future__ import print_function

import os
import matplotlib.pyplot as plt
import numpy as np

files = []

fig, ax = plt.subplots(figsize=(5,5))
for i in range(50):  # 50 frames
    plt.cla()
    plt.imshow(np.random.rand(5,5), interpolation='nearest')
    fname = '_tmp%03d.png%i' % (i)
    print('Saving frame', fname)
    plt.savefig(fname)
    files.append(fname)

print('Making movie animation.mpg - this make take a while')
#os.system("mencoder 'mf://_tmp*.png' -mf type=png:fps=10 -ovc lavc -lavcopts vcodec=wmv2 -oac copy -o animation.mpg")

# cleanup
for fname in files: os.remove(fname)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

from __future__ import print_function
from pylab import *
import matplotlib.cbook as cbook
# data are 256x256 16 bit integers
dfile = cbook.get_sample_data('s1045.ima.gz')
im = np.fromstring(dfile.read(), np.uint16).astype(float)
im.shape = 256, 256

#imshow(im, ColormapJet(256))
imshow(im, cmap=cm.gray)
axis('off')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
This now uses the imshow command instead of pcolor which *is much faster*
"""

from __future__ import division, print_function

import numpy as np

from matplotlib.pyplot import *
from matplotlib.collections import LineCollection
import matplotlib.cbook as cbook

# I use if 1 to break up the different regions of code visually

if 1:  # load the data
    # data are 256x256 16 bit integers
    dfile = cbook.get_sample_data('s1045.ima.gz')
    im = np.fromstring(dfile.read(), np.uint16).astype(float)
    im.shape = 256, 256

if 1:  # plot the MRI in pcolor
```python
subplot(221)
imshow(im, cmap=cm(gray)
axis('off')

if 1:
    # plot the histogram of MRI intensity
subplot(222)
 im = np.ravel(im)
 im = im[np.nonzero(im)] # ignore the background
 im = im/(2.0**15) # normalize
 hist(im, 100)
xticks([-1, -.5, 0, .5, 1])
yticks([])
xlabel('intensity')
ylabel('MRI density')

if 1:
    # plot the EEG
    # load the data
    numSamples, numRows = 800, 4
eegfile = cbook.get_sample_data('eeg.dat', asfileobj=False)
print('loading eeg %s' % eegfile)
data = np.fromstring(open(eegfile, 'rb').read(), float)
data.shape = numSamples, numRows
t = 10.0 * np.arange(numSamples, dtype=float)/numSamples
ticklocs = []
ax = subplot(212)
xlim(0,10)
xax = subplot(212)
xlim(0,10)
xticks(np.arange(10))
d = data.min()
d = data.min()
dmax = data.max()
dr = (dmax - dmin)*0.7 # Crowd them a bit.
y0 = dmin
y1 = (numRows-1) * dr + dmax
ylim(y0, y1)

segs = []
for i in range(numRows):
    segs.append(np.hstack((t[:,np.newaxis], data[:,i,np.newaxis],
    ticklocs.append(i*dr)
offsets = np.zeros((numRows,2), dtype=float)
offsets[:,1] = ticklocs

lines = LineCollection(segs, offsets=offsets,
    transOffset=None,)
ax.add_collection(lines)

# set the yticks to use axes coords on the y axis
ax.set_yticks(ticklocos)
ax.set_yticklabels(['PG3', 'PG5', 'PG7', 'PG9'])
```

Multiple images

#!/usr/bin/env python

Make a set of images with a single colormap, norm, and colorbar.

It also illustrates colorbar tick labelling with a multiplier.

from matplotlib.pyplot import figure, show, axes, sci
from matplotlib import cm, colors
from matplotlib.font_manager import FontProperties
from numpy import amin, amax, ravel
from numpy.random import rand

Nr = 3
Nc = 2

fig = figure()
cmap = cm.cool

figtitle = 'Multiple images'
t = fig.text(0.5, 0.95, figtitle,
    horizontalalignment='center',
    fontproperties=FontProperties(size=16))

cax = fig.add_axes([0.2, 0.08, 0.6, 0.04])

w = 0.4
h = 0.22
ax = []
images = []
vmin = 1e40
vmax = -1e40
for i in range(Nr):
    for j in range(Nc):
        pos = [0.075 + j*1.1*w, 0.18 + i*1.2*h, w, h]
        a = fig.add_axes(pos)
        if i > 0:
            a.set_xticklabels([])
        # Make some fake data with a range that varies
        # somewhat from one plot to the next.
        data = ((1+i+j)/10.0)*rand(10,20)*1e-6
        dd = ravel(data)
        # Manually find the min and max of all colors for
        # use in setting the color scale.
        vmin = min(vmin, amin(dd))
        vmax = max(vmax, amax(dd))
        images.append(a.imshow(data, cmap=cmap))
        ax.append(a)

# Set the first image as the master, with all the others
# observing it for changes in cmap or norm.

class ImageFollower:
    'update image in response to changes in clim or cmap on another image'
    def __init__(self, follower):
        self.follower = follower
    def __call__(self, leader):
        self.follower.set_cmap(leader.get_cmap())
        self.follower.set_clim(leader.get_clim())

norm = colors.Normalize(vmin=vmin, vmax=vmax)
for i, im in enumerate(images):
    im.set_norm(norm)
    if i > 0:
        images[0].callbacksSM.connect('changed', ImageFollower(im))
# The colorbar is also based on this master image.
fig.colorbar(images[0], cax, orientation='horizontal')

# We need the following only if we want to run this interactively and
# modify the colormap:

axes(ax[0])  # Return the current axes to the first one,
si(images[0])  # because the current image must be in current axes.

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.140  *pylab_examples*  example code: *multicolored_line.py*
#!/usr/bin/env python

"""
Color parts of a line based on its properties, e.g., slope.
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.colors import ListedColormap, BoundaryNorm

x = np.linspace(0, 3 * np.pi, 500)
y = np.sin(x)
z = np.cos(0.5 * (x[:-1] + x[1:]))  # first derivative

# Create a colormap for red, green and blue and a norm to color
# f < -0.5 red, f > 0.5 blue, and the rest green

cmap = ListedColormap(['r', 'g', 'b'])
norm = BoundaryNorm([-1, -0.5, 0.5, 1], cmap.N)

# Create a set of line segments so that we can color them individually
# This creates the points as a N x 1 x 2 array so that we can stack points
# together easily to get the segments. The segments array for line collection
# needs to be numlines x points per line x 2 (x and y)

points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)
# Create the line collection object, setting the colormapping parameters.  
# Have to set the actual values used for colormapping separately.
lc = LineCollection(segments, cmap=cmap, norm=norm)
lc.set_array(z)
lc.set_linewidth(3)

fig1 = plt.figure()
plt.gca().add_collection(lc)
plt.xlim(x.min(), x.max())
plt.ylim(-1.1, 1.1)

# Now do a second plot coloring the curve using a continuous colormap

t = np.linspace(0, 10, 200)
x = np.cos(np.pi * t)
y = np.sin(t)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

lc = LineCollection(segments, cmap=plt.get_cmap('copper'),
                    norm=plt.Normalize(0, 10))
lc.set_array(t)
lc.set_linewidth(3)

fig2 = plt.figure()
plt.gca().add_collection(lc)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
#!/usr/bin/env python
from pylab import *
# from matplotlib.pyplot import *
# from numpy import arange
if 1:
    figure(figsize=(7, 4))
    ax = subplot(121)
    ax.set_aspect(1)
    plot(arange(10))
    xlabel('this is a xlabel
    (with newlines!)')
    ylabel('this is vertical
test')
    text(2, 7, 'this is yet another test',
         rotation=45,
         horizontalalignment='center',
         verticalalignment='top',
         multialignment='center')
    grid(True)

subplot(122)

text(0.29, 0.7, 'Mat\n
TTp

123', size=18,
     va="baseline", ha="right", multialignment="left",
```

84.141 pylab_examples example code: multiline.py
84.142  pylab_examples example code: multipage_pdf.py

# This is a demo of creating a pdf file with several pages.

import datetime
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages
import matplotlib.pyplot as plt

# Create the PdfPages object to which we will save the pages:
# The with statement makes sure that the PdfPages object is closed properly at
# the end of the block, even if an Exception occurs.
with PdfPages('multipage_pdf.pdf') as pdf:
    plt.figure(figsize=(3, 3))
    plt.plot(range(7), [3, 1, 4, 1, 5, 9, 2], 'r-o')
    plt.title('Page One')
    pdf.savefig()  # saves the current figure into a pdf page
    plt.close()

    plt.rc('text', usetex=True)
    plt.figure(figsize=(8, 6))
    x = np.arange(0, 5, 0.1)
    plt.plot(x, np.sin(x), 'b-')
    plt.title('Page Two')
    pdf.savefig()
    plt.close()

    plt.rc('text', usetex=False)
    fig = plt.figure(figsize=(4, 5))
plt.plot(x, x**x, 'ko')
plt.title('Page Three')
pdf.savefig(fig)  # or you can pass a Figure object to pdf.savefig
plt.close()

# We can also set the file's metadata via the PdfPages object:
d = pdf.infodict()
d['Title'] = 'Multipage PDF Example'
d['Author'] = u'Jouni K. Seppänen'
d['Subject'] = 'How to create a multipage pdf file and set its metadata'
d['Keywords'] = 'PdfPages multipage keywords author title subject'
d['CreationDate'] = datetime.datetime(2009, 11, 13)
d['ModDate'] = datetime.datetime.today()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.143 pylab_examples example code: multiple_figs_demo.py
#!/usr/bin/env python
# Working with multiple figure windows and subplots
from pylab import *

# Create sine waves

t = arange(0.0, 2.0, 0.01)
s1 = sin(2*pi*t)
s2 = sin(4*pi*t)

# Create and populate two figures

figure(1)
s1.plot(t,s1)
s2.subplot(212)
s1.plot(t,2*s1)

figure(2)
s1.plot(t,s2)

# Now switch back to figure 1 and make some changes

figure(1)
s1.subplot(211)
s1.plot(t,s2)
s1.setp(gca(), 'xticklabels', [])

figure(1)
s1.savefig('fig1')
figure(2)
savefig('fig2')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.144 pylab_examples example code: multiple_yaxis_with_spines.py

import matplotlib.pyplot as plt

def make_patch_spines_invisible(ax):
    ax.set_frame_on(True)
    ax.patch.set_visible(False)
    for sp in ax.spines.itervalues():
        sp.set_visible(False)

fig, host = plt.subplots()
fig.subplots_adjust(right=0.75)
par1 = host.twinx()
par2 = host.twinx()

par1 = host.twinx()
par2 = host.twinx()
# Offset the right spine of par2. The ticks and label have already been
# placed on the right by twinx above.
par2.spines["right"].set_position(('axes', 1.2))
# Having been created by twinx, par2 has its frame off, so the line of its
# detached spine is invisible. First, activate the frame but make the patch
# and spines invisible.
make_patch_spines_invisible(par2)
# Second, show the right spine.
par2.spines["right"].set_visible(True)

p1, = host.plot([0, 1, 2], [0, 1, 2], "b-", label="Density")
p2, = par1.plot([0, 1, 2], [0, 3, 2], "r-", label="Temperature")
p3, = par2.plot([0, 1, 2], [50, 30, 15], "g-", label="Velocity")

host.set_xlim(0, 2)
host.set_ylim(0, 2)
par1.set_ylim(0, 4)
par2.set_ylim(1, 65)

host.set_xlabel("Distance")
host.set_ylabel("Density")
par1.set_ylabel("Temperature")
par2.set_ylabel("Velocity")

host.yaxis.label.set_color(p1.get_color())
par1.yaxis.label.set_color(p2.get_color())
par2.yaxis.label.set_color(p3.get_color())

tkw = dict(size=4, width=1.5)
host.tick_params(axis='y', colors=p1.get_color(), **tkw)
par1.tick_params(axis='y', colors=p2.get_color(), **tkw)
par2.tick_params(axis='y', colors=p3.get_color(), **tkw)
host.tick_params(axis='x', **tkw)

lines = [p1, p2, p3]

host.legend(lines, [l.get_label() for l in lines])

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
Example: simple line plots with NaNs inserted.
"""
from pylab import *
import numpy as np

# t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*2*pi*t)
t[41:60] = NaN  # t[50:60] = np.inf

subplot(2,1,1)
plot(t, s, '-*', lw=2)
xlabel('time (s)')
ylabel('voltage (mV)')
title('A sine wave with a gap of NaNs between 0.4 and 0.6')
grid(True)

subplot(2,1,2)
t[0] = NaN

more nans
Also with NaN in first and last point

A sine wave with a gap of NaNs between 0.4 and 0.6
t[-1] = NaN
plot(t, s, '-lw=2')
title('Also with NaN in first and last point')
xlabel('time (s)')
ylabel('more nans')
grid(True)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.146 pylab_examples example code: newscalarformatter_demo.py

The old formatter
The new formatter, default settings

![Graph showing the new formatter with default settings.](image)
The new formatter, no numerical offset

The new formatter, no numerical offset
#!/usr/bin/env python
# Demonstrating the improvements and options of the proposed new ScalarFormatter
from pylab import *
from matplotlib.ticker import OldScalarFormatter
x=frange(0,1,.01)
f=figure(figsize=(6,6))
f.text(0.5,0.975,'The old formatter',horizontalalignment='center',verticalalignment='top')
subplot(221)
plot(x*1e5+1e10,x*1e-10+1e-5)
gca().xaxis.set_major_formatter(OldScalarFormatter())
gca().yaxis.set_major_formatter(OldScalarFormatter())
subplot(222)
plot(x*1e5,x*1e-4)
gca().xaxis.set_major_formatter(OldScalarFormatter())
gca().yaxis.set_major_formatter(OldScalarFormatter())

Chapter 84. pylab_examples Examples
```python
subplot(223)
plot(-x*1e5-1e10,-x*1e-5-1e-10)
gca().xaxis.set_major_formatter(OldScalarFormatter())
gca().yaxis.set_major_formatter(OldScalarFormatter())
subplot(224)
plot(-x*1e5,-x*1e-4)
gca().xaxis.set_major_formatter(OldScalarFormatter())
gca().yaxis.set_major_formatter(OldScalarFormatter())

x=frange(0,1,.01)
f=figure(fignsize=(6,6))
f.text(0.5,0.975,"The new formatter, default settings",horizontalalignment='center',
verticalalignment='top')
subplot(221)
plot(x*1e5+1e10,x*1e-10+1e-5)
gca().xaxis.set_major_formatter(ScalarFormatter())
gca().yaxis.set_major_formatter(ScalarFormatter())
subplot(222)
plot(x*1e5,x*1e-4)
gca().xaxis.set_major_formatter(ScalarFormatter())
gca().yaxis.set_major_formatter(ScalarFormatter())
subplot(223)
plot(-x*1e5-1e10,-x*1e-5-1e-10)
gca().xaxis.set_major_formatter(ScalarFormatter())
gca().yaxis.set_major_formatter(ScalarFormatter())
subplot(224)
plot(-x*1e5,-x*1e-4)
gca().xaxis.set_major_formatter(ScalarFormatter())
gca().yaxis.set_major_formatter(ScalarFormatter())

x=frange(0,1,.01)
f=figure(fignsize=(6,6))
f.text(0.5,0.975,"The new formatter, no numerical offset",horizontalalignment='center',
verticalalignment='top')
subplot(221)
plot(x*1e5+1e10,x*1e-10+1e-5)
gca().xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
gca().yaxis.set_major_formatter(ScalarFormatter(useOffset=False))
subplot(222)
plot(x*1e5,x*1e-4)
gca().xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
gca().yaxis.set_major_formatter(ScalarFormatter(useOffset=False))
subplot(223)
plot(-x*1e5-1e10,-x*1e-5-1e-10)
gca().xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
gca().yaxis.set_major_formatter(ScalarFormatter(useOffset=False))
subplot(224)
plot(-x*1e5,-x*1e-4)
gca().xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
gca().yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

x=frange(0,1,.01)
f=figure(fignsize=(6,6))
```
f.text(0.5, 0.975, 'The new formatter, with mathtext', horizontalalignment='center', verticalalignment='top')

subplot(221)
plot(x*1e5+1e10, x*1e-10+1e-5)
gca().xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
gca().yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

subplot(222)
plot(x*1e5, x*1e-4)
gca().xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
gca().yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

subplot(223)
plot(-x*1e5-1e10, -x*1e-5-1e-10)
gca().xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
gca().yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

subplot(224)
plot(-x*1e5, -x*1e-4)
gca().xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
gca().yaxis.set_major_formatter(ScalarFormatter(useMathText=True))
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.147  pylab_examples example code: patheffect_demo.py

```python
import matplotlib.pyplot as plt
import matplotlib.patheffects as PathEffects
import numpy as np

if 1:
    plt.figure(1, figsize=(8, 3))
    ax1 = plt.subplot(131)
    ax1.imshow([[1, 2], [2, 3]])
    txt = ax1.annotate("test", (1., 1.), (0., 0),
                     arrowprops=dict(arrowstyle="->",
                                     connectionstyle="angle3", lw=2),
```

2220 Chapter 84. pylab_examples Examples
size=20, ha="center", path_effects=[PathEffects.withStroke(linewidth=3, foreground="w")])

txt.arrow_patch.set_path_effects([PathEffects.Stroke(linewidth=5, foreground="w"), PathEffects.Normal()])

ax1.grid(True, linestyle="-")

pe = [PathEffects.withStroke(linewidth=3, foreground="w")]
for l in ax1.get_xgridlines() + ax1.get_ygridlines():
    l.set_path_effects(pe)

ax2 = plt.subplot(132)
arr = np.arange(25).reshape((5, 5))
ax2.imshow(arr)
cntr = ax2.contour(arr, colors="k")
plt.setp(cntr.collections, path_effects=[PathEffects.withStroke(linewidth=3, foreground="w")])
clbls = ax2.clabel(cntr, fmt="%2.0f", use_clabeltext=True)
plt.setp(clbls, path_effects=[PathEffects.withStroke(linewidth=3, foreground="w")])

# shadow as a path effect
ax3 = plt.subplot(133)
p1, = ax3.plot([0, 1], [0, 1])
leg = ax3.legend([p1], ["Line 1"], fancybox=True, loc=2)
leg.legendPatch.set_path_effects([PathEffects.withSimplePatchShadow()])

plt.show()
Demonstrates similarities between pcolor, pcolormesh, imshow and pcolorfast for drawing quadrilateral grids.

```python
import matplotlib.pyplot as plt
import numpy as np

# make these smaller to increase the resolution
dx, dy = 0.15, 0.05

# generate 2d grids for the x & y bounds
y, x = np.mgrid[slice(-3, 3 + dy, dy),
                slice(-3, 3 + dx, dx)]
z = (1 - x / 2. + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)
# x and y are bounds, so z should be the value **inside** those bounds.
# Therefore, remove the last value from the z array.
z = z[:-1, :-1]
z_min, z_max = -np.abs(z).max(), np.abs(z).max()
```

2222 Chapter 84. pylab_examples Examples
```
plt.subplot(2, 2, 1)
plt.pcolor(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolor')
# set the limits of the plot to the limits of the data
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.colorbar()

plt.subplot(2, 2, 2)
plt.pcolormesh(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolormesh')
# set the limits of the plot to the limits of the data
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.colorbar()

plt.subplot(2, 2, 3)
plt.imshow(z, cmap='RdBu', vmin=z_min, vmax=z_max,
          extent=[x.min(), x.max(), y.min(), y.max()],
          interpolation='nearest', origin='lower')
plt.title('image (interp. nearest)')
plt.colorbar()

ax = plt.subplot(2, 2, 4)
ax.pcolorfast(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolorfast')
plt.colorbar()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
```python
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import numpy as np
from matplotlib.mlab import bivariate_normal

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) + 0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)

plt.subplot(2,1,1)
plt.pcolor(X, Y, Z1, norm=LogNorm(vmin=Z1.min(), vmax=Z1.max()), cmap='PuBu_r')
plt.colorbar()

plt.subplot(2,1,2)
plt.pcolor(X, Y, Z1, cmap='PuBu_r')
plt.colorbar()
```
import matplotlib.pyplot as plt
from numpy.random import rand

Z = rand(6,10)

c = plt.pcolor(Z)
c = plt.pcolor(Z, edgecolors='k', linewidths=4)

plt.show()
Make a pie charts of varying size – see http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.pie for the docstring.

This example shows a basic pie charts with labels optional features, like autolabeling the percentage, offsetting a slice with "explode" and adding a shadow, in different sizes.

```python
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

# Some data
labels = ['Frogs', 'Hogs', 'Dogs', 'Logs']
fracs = [15, 30, 45, 10]
explode=(0, 0.05, 0, 0)

# Make square figures and axes
the_grid = GridSpec(2, 2)
```
plt.subplot(the_grid[0, 0], aspect=1)
plt.pie(fracs, labels=labels, autopct='%1.1f%%', shadow=True)
plt.subplot(the_grid[0, 1], aspect=1)
plt.pie(fracs, explode=explode, labels=labels, autopct='%.0f%%', shadow=True)
plt.subplot(the_grid[1, 0], aspect=1)
patches, texts, autotexts = plt.pie(fracs, labels=labels,
    autopct='%.0f%%',
    shadow=True, radius=0.5)

# Make the labels on the small plot easier to read.
for t in texts:
    t.set_size('smaller')
for t in autotexts:
    t.set_size('x-small')
autotexts[0].set_color('y')

plt.subplot(the_grid[1, 1], aspect=1)
patches, texts, autotexts = plt.pie(fracs, explode=explode,
    labels=labels, autopct='%.0f%%',
    shadow=False, radius=0.5)
# Turn off shadow for tiny plot
# with exploded slice.
for t in texts:
    t.set_size('smaller')
for t in autotexts:
    t.set_size('x-small')
autotexts[0].set_color('y')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.152 `pylab_examples` example code: `plotfile_demo.py`
$f(x) = x^2$

$y = x^3$

---

84.152. `pylab_examples example code: plotfile_demo.py`
$f(x) = x^2, x^3$
from pylab import plotfile, show, gca
import matplotlib.cbook as cbook

fname = cbook.get_sample_data('msft.csv', asfileobj=False)
fname2 = cbook.get_sample_data('data_x_x2_x3.csv', asfileobj=False)

# test 1; use ints
plotfile(fname, (0,5,6))

# test 2; use names
plotfile(fname, ('date', 'volume', 'adj_close'))

# test 3; use semilogy for volume
plotfile(fname, ('date', 'volume', 'adj_close'), plotfuncs={'volume': 'semilogy'})

# test 4; use semilogy for volume
plotfile(fname, (0,5,6), plotfuncs={5:'semilogy'})

#test 5; single subplot
plotfile(fname, ('date', 'open', 'high', 'low', 'close'), subplots=False)

# test 6; labeling, if no names in csv-file
plotfile(fname2, cols=(0,1,2), delimiter=' ',
    names=['$x$', '$f(x)=x^2$', '$f(x)=x^3$'])
# test 7; more than one file per figure--illustrated here with a single file
plotfile(fname2, cols=(0, 1), delimiter=' ')  
plotfile(fname2, cols=(0, 2), newfig=False, delimiter=' ') # use current figure

gca().set_xlabel(r'$x$')
gca().set_ylabel(r'$f(x) = x^2, x^3$')

# test 8; use bar for volume
plotfile(fname, (0,5,6), plotfuncs={5: 'bar'})

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.153 pylab_examples example code: polar_demo.py

A line plot on a polar axis

Demo of a line plot on a polar axis.

import numpy as np
import matplotlib.pyplot as plt
r = np.arange(0, 3.0, 0.01)
theta = 2 * np.pi * r

ax = plt.subplot(111, polar=True)
ax.plot(theta, r, color='r', linewidth=3)
ax.set_rmax(2.0)
ax.grid(True)

ax.set_title("A line plot on a polar axis", va='bottom')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

import numpy as np
from matplotlib.pyplot import figure, show, rc

# radar green, solid grid lines
rc('grid', color='#316931', linewidth=1, linestyle='--')
rc('xtick', labelsize=15)
rc('ytick', labelsize=15)

# force square figure and square axes looks better for polar, IMO
```python
fig = figure(figsize=(8, 8))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True, axisbg='#d5de9c')

r = np.arange(0, 3.0, 0.01)
theta = 2*np.pi*r
ax.plot(theta, r, color='#ee8d18', lw=3, label='a line')
ax.plot(0.5*theta, r, color='blue', ls='--', lw=3, label='another line')
ax.legend()

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 84.155  `pylab_examples` example code: `print_stdout.py`

[source code]

```python
#!/usr/bin/env python
# -*- noplot -*-
# print png to standard out
# usage: python print_stdout.py > somefile.png
import sys
import matplotlib
matplotlib.use('Agg')
from pylab import *

plot([1,2,3])

savefig(sys.stdout)
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
# python

from pylab import *

dt = 0.01
t = arange(0,10,dt)
nse = randn(len(t))
r = exp(-t/0.05)

cnse = convolve(nse, r)*dt
cnse = cnse[:len(t)]
s = 0.1*sin(2*pi*t) + cnse

subplot(211)
plot(t,s)
subplot(212)
psd(s, 512, 1/dt)

show()
dt = 0.01;
t = [0:dt:10];
nse = randn(size(t));
r = exp(-t/0.05);
cnse = conv(nse, r)*dt;
cnse = cnse(1:length(t));
s = 0.1*sin(2*pi*t) + cnse;

subplot(211)
plot(t,s)
subplot(212)
psd(s, 512, 1/dt)

#This example shows the effects of some of the different PSD parameters
import numpy as np
import matplotlib.pyplot as plt
```
dt = np.pi / 100.
fs = 1. / dt
t = np.arange(0, 8, dt)
y = 10. * np.sin(2 * np.pi * 4 * t) + 5. * np.sin(2 * np.pi * 4.25 * t)
y = y + np.random.randn(*t.shape)

#Plot the raw time series
fig = plt.figure()
fig.subplots_adjust(hspace=0.45, wspace=0.3)
ax = fig.add_subplot(2, 1, 1)
ax.plot(t, y)

#Plot the PSD with different amounts of zero padding. This uses the entire
time series at once
ax2 = fig.add_subplot(2, 3, 4)
ax2.psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
ax2.psd(y, NFFT=len(t), pad_to=len(t)*2, Fs=fs)
ax2.psd(y, NFFT=len(t), pad_to=len(t)*4, Fs=fs)
plt.title('zero padding')

#Plot the PSD with different block sizes, Zero pad to the length of the original
data sequence.
ax3 = fig.add_subplot(2, 3, 5, sharex=ax2, sharey=ax2)
ax3.psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
ax3.psd(y, NFFT=len(t)//2, pad_to=len(t), Fs=fs)
ax3.psd(y, NFFT=len(t)//4, pad_to=len(t), Fs=fs)
ax3.set_ylabel('')
plt.title('block size')

#Plot the PSD with different amounts of overlap between blocks
ax4 = fig.add_subplot(2, 3, 6, sharex=ax2, sharey=ax2)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=0, Fs=fs)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=int(0.05*len(t)/2.), Fs=fs)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=int(0.2*len(t)/2.), Fs=fs)
ax4.set_ylabel('')
plt.title('overlap')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#This is a ported version of a MATLAB example from the signal processing toolbox that showed some difference at one time between Matplotlib’s and MATLAB’s scaling of the PSD.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

fs = 1000
f = np.linspace(0, 0.3, 301)
A = np.array([2, 8]).reshape(-1, 1)
f = np.array([150, 140]).reshape(-1, 1)
xn = (A * np.sin(2 * np.pi * f * t)).sum(axis=0) + 5 * np.random.randn(*t.shape)

plt.subplots_adjust(hspace=0.45, wspace=0.3)
plt.subplot(1, 2, 1)
plt.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024, scale_by_freq=True)
plt.title('Periodogram')

plt.subplot(1, 2, 2)
plt.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024, scale_by_freq=True)
plt.title('Welch')
```
```python
plt.yticks(yticks)
plt.xticks(xticks)
plt.grid(True)

plt.subplot(1,2,2)
plt.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, noverlap=75, pad_to=512,
       scale_by_freq=True)
plt.title('Welch')
plt.xticks(xticks)
plt.yticks(yticks)
plt.ylabel('')
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

---

#This is a ported version of a MATLAB example from the signal processing toolbox that showed some difference at one time between Matplotlib's and MATLAB's scaling of the PSD. This differs from psd_demo3.py in that
# this uses a complex signal, so we can see that complex PSD's work properly
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

fs = 1000
f = np.linspace(0, 0.3, 301)
A = np.array([2, 8]).reshape(-1, 1)
f = np.array([150, 140]).reshape(-1, 1)
xn = (A * np.exp(2j * np.pi * f * t)).sum(axis=0) + 5 * np.random.randn(*t.shape)

plt.subplots_adjust(hspace=0.45, wspace=0.3)
ax = plt.subplot(1, 2, 1)
plt.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024, scale_by_freq=True)
plt.title('Periodogram')
plt.yticks(yticks)
plt.xticks(xticks)
plt.grid(True)
plt.xlim(-500, 500)

plt.subplot(1, 2, 2, sharex=ax, sharey=ax)
plt.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, noverlap=75, pad_to=512, scale_by_freq=True)
plt.title('Welch')
plt.yticks(yticks)
plt.xticks(xticks)
plt.yticks(yticks)
plt.ylabel('')
plt.grid(True)
plt.xlim(-500, 500)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.160  pylab_examples example code: ptest.py

```
#!/usr/bin/env python
# -*- noplot -*-
import matplotlib
matplotlib.use('PS')
from pylab import *

def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
```
return multiply(s1,e1)

t1 = arange(0.0, 5.0, .1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

figure(1)
subplot(211)
l = plot(t1, f(t1), 'k^')
setp(l, markerfacecolor='k', markeredgecolor='r')
title('A tale of 2 subplots', fontsize=14, fontname='Courier')
ylabel('Signal 1', fontsize=12)

subplot(212)
l = plot(t1, f(t1), 'k>')
ylabel('Signal 2', fontsize=12)
xlabel('time (s)', fontsize=12)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
Some people prefer to write more pythonic, object oriented, code rather than use the pylab interface to matplotlib. This example shows you how.

Unless you are an application developer, I recommend using part of the pylab interface, particularly the figure, close, subplot, axes, and show commands. These hide a lot of complexity from you that you don’t need to see in normal figure creation, like instantiating DPI instances, managing the bounding boxes of the figure elements, creating and reasizing GUI windows and embedding figures in them.

If you are an application developer and want to embed matplotlib in your application, follow the lead of examples/embedding_in_wx.py, examples/embedding_in_gtk.py or examples/embedding_in_tk.py. In this case you will want to control the creation of all your figures, embedding them in application windows, etc.

If you are a web application developer, you may want to use the example in webapp_demo.py, which shows how to use the backend agg
figure can be done directly, with none of the globals (current figure, current axes) that are present in the *pylab* interface. Note that there is no reason why the *pylab* interface won’t work for web application developers, however.

If you see an example in the examples dir written in *pylab* interface, and you want to emulate that using the true python method calls, there is an easy mapping. Many of those examples use 'set' to control figure properties. Here's how to map those commands onto instance methods

The syntax of set is

```python
setp(object or sequence, somestring, attribute)
```

if called with an object, set calls

```python
object.set_somestring(attribute)
```

if called with a sequence, set does

```python
for object in sequence:
    object.set_somestring(attribute)
```

So for your example, if a is your axes object, you can do

```python
a.set_xticklabels([])
a.set_yticklabels([])
a.set_xticks([])
a.set_yticks([])
```

```python
from pylab import figure, show
from numpy import arange, sin, pi

t = arange(0.0, 1.0, 0.01)

fig = figure()

ax1 = fig.add_subplot(211)
ax1.plot(t, sin(2*pi*t))
ax1.grid(True)
ax1.set_ylim(( -2, 2 ) )
ax1.set_ylabel( '1 Hz' )
ax1.set_title( 'A sine wave or two' )

for label in ax1.get_xticklabels():
    label.set_color('r')

ax2 = fig.add_subplot(212)
ax2.plot(t, sin(2*2*pi*t))
```
ax2.grid(True)
ax2.set_ymlim((-2,2))
l = ax2.set_xlabel('Hi mom')
l.set_color('g')
l.set_fontsize('large')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.162 pylab_examples example code: quadmesh_demo.py

```python
#!/usr/bin/env python

""
pcolormesh uses a QuadMesh, a faster generalization of pcolor, but with some restrictions.

This demo illustrates a bug in quadmesh with masked data.
""

import numpy as np
from matplotlib.pyplot import figure, show, savefig
```

84.162. pylab_examples example code: quadmesh_demo.py
from matplotlib import cm, colors
from numpy import ma

n = 12
x = np.linspace(-1.5, 1.5, n)
y = np.linspace(-1.5, 1.5, n*2)
X, Y = np.meshgrid(x, y);
Qx = np.cos(Y) - np.cos(X)
Qz = np.sin(Y) + np.sin(X)
Qx = (Qx + 1.1)
Z = np.sqrt(X**2 + Y**2)/5;
Z = (Z - Z.min()) / (Z.max() - Z.min())

# The color array can include masked values:
Zm = ma.masked_where(np.fabs(Qz) < 0.5*amax(Qz), Z)

fig = figure()
ax = fig.add_subplot(121)
ax.set_axis_bgcolor("#bdb76b")
ax.pcolormesh(Qx, Qz, Z, shading='gouraud')
ax.set_title('Without masked values')

ax = fig.add_subplot(122)
ax.set_axis_bgcolor("#bdb76b")
# You can control the color of the masked region:
#cmap = cm.jet
#cmap.set_bad('r', 1.0)
#ax.pcolormesh(Qx, Qz, Zm, cmap=cmap)
# Or use the default, which is transparent:
col = ax.pcolormesh(Qx, Qz, Zm, shading='gouraud')
ax.set_title('With masked values')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Minimal arguments, no kwargs

Minimal arguments, no kwargs
scales with plot width, not view

\[ 2^{\frac{m}{3}} \]
pivot='mid'; every third arrow; units='inches'
scales with x view; pivot='tip' - $\frac{1}{x}$
triangular head; scale with x view; black edges
Demonstration of quiver and quiverkey functions. This is using the new version coming from the code in quiver.py.

Known problem: the plot autoscaling does not take into account the arrows, so those on the boundaries are often out of the picture. This is *not* an easy problem to solve in a perfectly general way. The workaround is to manually expand the axes.

```python
from pylab import *
from numpy import ma

X, Y = meshgrid( arange(0,2*pi,.2), arange(0,2*pi,.2) )
U = cos(X)
V = sin(Y)

#1
figure()
Q = quiver(U, V)
qk = quiverkey(Q, 0.5, 0.92, 2, r'$2 \frac{m}{s}$', labelpos='W',
              fontproperties={'weight': 'bold'})
l, r, b, t = axis()
dx, dy = r-l, t-b
axis([l-0.05*dx, r+0.05*dx, b-0.05*dy, t+0.05*dy])
```
Minimal arguments, no kwargs

```
#2
figure()
Q = quiver(X, Y, U, V, units='width')
qk = quiverkey(Q, 0.9, 0.95, 2, r'$2 \frac{m}{s}$',
               labelpos='E',
               coordinates='figure',
               fontproperties={'weight': 'bold'})
axis([-1, 7, -1, 7])
title('Minimal arguments, no kwargs')

#3
figure()
Q = quiver(X[::-3, ::3], Y[::-3, ::3], U[::-3, ::3], V[::-3, ::3],
pivot='mid', color='r', units='inches')
qk = quiverkey(Q, 0.5, 0.03, 1, r'$1 \frac{m}{s}$',
               fontproperties={'weight': 'bold'})
plot(X[::-3, ::3], Y[::-3, ::3], 'k')
axis([-1, 7, -1, 7])
title('pivot=mid; every third arrow; units=inches')

#4
figure()
M = sqrt(pow(U, 2) + pow(V, 2))
Q = quiver(X, Y, U, V, M, units='x', pivot='tip', width=0.022, scale=1/0.15)
qk = quiverkey(Q, 0.9, 0.92, 1, r'$1 \frac{m}{s}$',
               labelpos='W',
               fontproperties={'weight': 'bold'})
plot(X, Y, 'k')
axis([-1, 7, -1, 7])
title('triangular head; scale with x view; black edges')

#5
figure()
M = zeros(U.shape, dtype=bool)
U = ma.masked_array(U, mask=M)
V = ma.masked_array(V, mask=M)
Q = quiver(U, V)
qk = quiverkey(Q, 0.5, 0.92, 2, r'$2 \frac{m}{s}$',
               labelpos='W',
               fontproperties={'weight': 'bold'})
```

84.163. pylab_examples example code: quiver_demo.py
title('Minimal arguments, no kwargs - masked values')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.164 pylab_examples example code: scatter_custom_symbol.py

import matplotlib.pyplot as plt
from numpy import arange, pi, cos, sin
from numpy.random import rand

# unit area ellipse
rx, ry = 3., 1.
area = rx * ry * pi
theta = arange(0, 2*pi+0.01, 0.1)
verts = list(zip(rx/area*cos(theta), ry/area*sin(theta)))
x,y,s,c = rand(4, 30)
s**2 = 10**2.
```python
fig, ax = plt.subplots()
ax.scatter(x, y, s, c, marker=None, verts=verts)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.165  *pylab_examples* example code: *scatter_demo2.py*

```python
""
Demo of scatter plot with varying marker colors and sizes.
""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

# Load a numpy record array from yahoo csv data with fields date,
# open, close, volume, adj_close from the mpl-data/example directory.
# The record array stores python datetime.date as an object array in
# the date column
datafile = cbook.get_sample_data('goog.npy')
price_data = np.load(datafile).view(np.recarray)
```
price_data = price_data[-250:] # get the most recent 250 trading days

delta1 = np.diff(price_data.adj_close)/price_data.adj_close[:-1]

# Marker size in units of points^2
volume = (15 * price_data.volume[:-2] / price_data.volume[0])**2
close = 0.003 * price_data.close[:-2] / 0.003 * price_data.open[:-2]

fig, ax = plt.subplots()
ax.scatter(delta1[:-1], delta1[1:], c=close, s=volume, alpha=0.5)

ax.set_xlabel(r'$\Delta_i$', fontsize=20)
ax.set_ylabel(r'$\Delta_{i+1}$', fontsize=20)
ax.set_title('Volume and percent change')

ax.grid(True)
fig.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import NullFormatter

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)

nullfmt = NullFormatter()  # no labels

# definitions for the axes
left, width = 0.1, 0.65
bottom, height = 0.1, 0.65
bottom_h = left_h = left+width+0.02
rect_scatter = [left, bottom, width, height]
rect_histx = [left, bottom_h, width, 0.2]
rect_histy = [left_h, bottom, 0.2, height]

# start with a rectangular Figure
plt.figure(1, figsize=(8,8))

axScatter = plt.axes(rect_scatter)
axHistx = plt.axes(rect_histx)
axHisty = plt.axes(rect_histy)

# no labels
axHistx.xaxis.set_major_formatter(nullfmt)
axHisty.yaxis.set_major_formatter(nullfmt)

# the scatter plot:
axScatter.scatter(x, y)

# now determine nice limits by hand:
binwidth = 0.25
xmax = np.max( [np.max(np.fabs(x)), np.max(np.fabs(y))] )
lim = ( int(xmax/binwidth) + 1 ) * binwidth

axScatter.set_xlim( (-lim, lim) )
aXScatter.set_ylim( (-lim, lim) )

bins = np.arange(-lim, lim + binwidth, binwidth)
aXHistx.hist(x, bins=bins)
aXHisty.hist(y, bins=bins, orientation='horizontal')

axHistx.set_xlim( axScatter.get_xlim() )
aXHisty.set_ylim( axScatter.get_ylim() )

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
from pylab import *

N = 100
r0 = 0.6
x = 0.9*rand(N)
y = 0.9*rand(N)
area = pi*(10 * rand(N))**2 # 0 to 10 point radiuses
c = sqrt(area)
r = sqrt(x*x+y*y)
area1 = ma.masked_where(r < r0, area)
area2 = ma.masked_where(r >= r0, area)
scatter(x,y,s=area1, marker='^', c=c, hold='on')
scatter(x,y,s=area2, marker='o', c=c)

# Show the boundary between the regions:
theta = arange(0, pi/2, 0.01)
plot(r0*cos(theta), r0*sin(theta))

show()
```python
#!/usr/bin/env python
# -*- noplot -*-

import pylab
import time

for N in (20,100,1000,10000,50000):
    tstart = time.time()
    x = 0.9*pylab.rand(N)
    y = 0.9*pylab.rand(N)
    s = 20*pylab.rand(N)
    pylab.scatter(x,y,s)
    print('%d symbols in %1.2f s' % (N, time.time()-tstart))
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt

x = np.random.rand(10)
y = np.random.rand(10)
z = np.sqrt(x**2 + y**2)

plt.subplot(321)
plt.scatter(x,y,s=80, c=z, marker=">")

plt.subplot(322)
plt.scatter(x,y,s=80, c=z, marker=(5,0))
verts = list(zip([-1.,1.,1.,-1.],[-1.,-1.,1.,-1.]))
plt.subplot(323)
plt.scatter(x,y,s=80, c=z, marker=(verts,0))
# equivalent:
# plt.scatter(x,y,s=80, c=z, marker=None, verts=verts)

plt.subplot(324)
plt.scatter(x,y,s=80, c=z, marker=(5,1))
```python
from matplotlib import pyplot as plt
import numpy as np
import matplotlib

x = np.arange(0.0, 50.0, 2.0)
y = x ** 1.3 + np.random.rand(*x.shape) * 30.0
s = np.random.rand(*x.shape) * 800 + 500
plt.scatter(x, y, s, c="g", alpha=0.5, marker=r'\clubsuit', label="Luck")
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
plt.xlabel("Leprechauns")
plt.ylabel("Gold")
plt.legend(loc=2)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.171  pylab_examples example code: set_and_get.py

MATLAB and pylab allow you to use setp and get to set and get object properties, as well as to do introspection on the object

set

To set the linestyle of a line to be dashed, you can do

```python
>>> line, = plot([1,2,3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value
```python
>>> setp(line, 'linestyle')
linestyle: [ '-.' | '-.' | '-' | ':' | 'steps' | 'None' ]

If you want to see all the properties that can be set, and their possible values, you can do

```python
gt  setp(line)
```  
set operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red

```python
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

get:

```python
>>> getp(line, 'linewidth')
0.5
```

or all the attribute/value pairs

```python
>>> getp(line)
  aa = True
  alpha = 1.0
  antialiased = True
  c = b
  clip_on = True
  color = b
  ... long listing skipped ...
```

**Aliases:**

To reduce keystrokes in interactive mode, a number of properties have short aliases, e.g., 'lw' for 'linewidth' and 'mec' for 'markeredgecolor'. When calling set or get in introspection mode, these properties will be listed as 'fullname or aliasname', as in

```python
from __future__ import print_function
```
from pylab import *

x = arange(0, 1.0, 0.01)
y1 = sin(2*pi*x)
y2 = sin(4*pi*x)
lines = plot(x, y1, x, y2)
l1, l2 = lines
setp(lines, linestyle='--')  # set both to dashed
setp(l1, linewidth=2, color='r')  # line1 is thick and red
setp(l2, linewidth=1, color='g')  # line2 is thicker and green

print ('Line setters')
setp(l1)
print ('Line getters')
getp(l1)

print ('Rectangle setters')
setp(gca().patch)
print ('Rectangle getters')
getp(gca().patch)

t = title('Hi mom')
print ('Text setters')
setp(t)
print ('Text getters')
getp(t)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.172  pylab_examples example code: shading_example.py

![imshow](image1.png) ![imshow with shading](image2.png)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LightSource

# example showing how to make shaded relief plots
# like Mathematica
# (http://reference.wolfram.com/mathematica/ref/ReliefPlot.html)
# or Generic Mapping Tools

# test data
X, Y = np.mgrid[-5:5:0.05, -5:5:0.05]
Z = np.sqrt(X**2 + Y**2) + np.sin(X**2 + Y**2)

# create light source object.
ls = LightSource(azdeg=0, altdeg=65)

# shade data, creating an rgb array.
rgb = ls.shade(Z, plt.cm.copper)

# plot un-shaded and shaded images.
plt.figure(figsize=(12, 5))
plt.subplot(121)
plt.imshow(Z, cmap=plt.cm.copper)
plt.title('imshow')
plt.xticks([]); plt.yticks([])
plt.subplot(122)
plt.imshow(rgb)
plt.title('imshow with shading')
plt.xticks([]); plt.yticks([])
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.173 pylab_examples example code: shared_axis_across_figures.py

[source code]

"""
connect the data limits on the axes in one figure with the axes in
another. This is not the right way to do this for two axes in the
same figure -- use the sharex and sharey property in that case
"""

# -*- noplot -*-
import numpy
from pylab import figure, show

fig1 = figure()
fig2 = figure()

ax1 = fig1.add_subplot(111)
ax2 = fig2.add_subplot(111, sharex=ax1, sharey=ax1)
ax1.plot(numpy.random.rand(100), 'o')
a2.plot(numpy.random.rand(100), 'o')

# In the latest release, it is no longer necessary to do anything
# special to share axes across figures:

# ax1.sharex_foreign(a2)
# ax2.sharex_foreign(ax1)

# ax1.sharey_foreign(ax2)
# ax2.sharey_foreign(ax1)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.174  pylab_examples example code: shared_axis_demo.py

You can share the x or y axis limits for one axis with another by passing an axes instance as a sharex or sharey kwarg.
Changing the axis limits on one axes will be reflected automatically in the other, and vice-versa, so when you navigate with the toolbar the axes will follow each other on their shared axes. Ditto for changes in the axis scaling (e.g., log vs linear). However, it is possible to have differences in tick labeling, e.g., you can selectively turn off the tick labels on one axes.

The example below shows how to customize the tick labels on the various axes. Shared axes share the tick locator, tick formatter, view limits, and transformation (e.g., log, linear). But the ticklabels themselves do not share properties. This is a feature and not a bug, because you may want to make the tick labels smaller on the upper axes, e.g., in the example below.

If you want to turn off the ticklabels for a given axes (e.g., on subplot(211) or subplot(212), you cannot do the standard trick

```python
setp(ax2, xticklabels=[])  
```

because this changes the tick Formatter, which is shared among all axes. But you can alter the visibility of the labels, which is a property

```python
setp(ax2.get_xticklabels(), visible=False)
```

""

```python
from pylab import *

t = arange(0.01, 5.0, 0.01)
s1 = sin(2*pi*t)
s2 = exp(-t)
s3 = sin(4*pi*t)
ax1 = subplot(311)
plot(t, s1)
setp(ax1.get_xticklabels(), fontsize=6)
## share x only
ax2 = subplot(312, sharex=ax1)
plot(t, s2)
# make these tick labels invisible
setp(ax2.get_xticklabels(), visible=False)

# share x and y
ax3 = subplot(313, sharex=ax1, sharey=ax1)
plot(t, s3)
xlim(0.01, 5.0)
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
from pylab import *

t = arange(0.0, 2.0, 0.01)
s = sin(2*pi*t)
plot(t, s)
xlabel('time (s)')
ylabel('voltage (mV)')
title('About as simple as it gets, folks')
grid(True)
savefig('test.png')
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

Example: simple line plot.
Show how to make and save a simple line plot with labels, title and grid

# -*- noplot -*-
from __future__ import print_function
from pylab import *
ion()
t = arange(0.0, 1.0+0.001, 0.001)
s = cos(2*2*pi*t)
plot(t, s, '-.', lw=2)
xlabel('time (s)')
ylabel('voltage (mV)')
title('About as simple as it gets, folks')
grid(True)

import time
frames = 100.0
t = time.time()
c = time.clock()
for i in range(int(frames)):
    part = i / frames
    axis([0.0, 1.0 - part, -1.0 + part, 1.0 - part])
wallclock = time.time() - t
user = time.clock() - c
print("wallclock:", wallclock)
print("user:", user)
print("fps:", frames / wallclock)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
#!/usr/bin/env python
from pylab import *

dt = 0.0005
t = arange(0.0, 20.0, dt)
s1 = sin(2*pi*100*t)
s2 = 2*sin(2*pi*400*t)

# create a transient "chirp"
mask = where(logical_and(t>10, t<12), 1.0, 0.0)
s2 = s2 * mask

# add some noise into the mix
nse = 0.01*randn(len(t))
x = s1 + s2 + nse # the signal
NFFT = 1024 # the length of the windowing segments
Fs = int(1.0/dt) # the sampling frequency

# Pxx is the segments x freqs array of instantaneous power, freqs is # the frequency vector, bins are the centers of the time bins in which # the power is computed, and im is the matplotlib.image.AxesImage
```

84.177. `pylab_examples` example code: `specgram_demo.py`
# instance

ax1 = subplot(211)
plot(t, x)
subplot(212, sharex=ax1)

Pxx, freqs, bins, im = specgram(x, NFFT=NFFT, Fs=Fs, noverlap=900,
cmap=cm.gist_heat)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.178 pylab_examples example code: spectrum_demo.py

```python
#!/usr/bin/env python
# python

from pylab import *

dt = 0.01
Fs = 1/dt
t = arange(0, 10, dt)
nse = randn(len(t))
nse = randn(len(t))
```
```python
r = exp(-t/0.05)
cnse = convolve(nse, r)*dt
cnse = cnse[:len(t)]
s = 0.1*sin(2*pi*t) + cnse

subplot(3, 2, 1)
plot(t, s)

subplot(3, 2, 3)
magnitude_spectrum(s, Fs=Fs)

subplot(3, 2, 4)
magnitude_spectrum(s, Fs=Fs, scale='dB')

subplot(3, 2, 5)
angle_spectrum(s, Fs=Fs)

subplot(3, 2, 6)
phase_spectrum(s, Fs=Fs)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
centered spines

spines at axes (0.6, 0.1)

zeroed spines

spines at data (1, 2)
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
x = np.linspace(-np.pi, np.pi, 100)
y = 2*np.sin(x)

ax = fig.add_subplot(2, 2, 1)
ax.set_title('centered spines')
ax.plot(x, y)
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 2)
ax.set_title('zeroed spines')
ax.plot(x, y)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 3)
ax.set_title('spines at axes (0.6, 0.1)')
ax.plot(x, y)
ax.spines['left'].set_position(('axes', 0.6))
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position(('axes', 0.1))
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 4)
ax.set_title('spines at data (1, 2)')
ax.plot(x, y)
ax.spines['left'].set_position(('data', 1))
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position(('data', 2))
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

# ----------------------------------------------------

def adjust_spines(ax, spines):
    for loc, spine in ax.spines.items():
        if loc in spines:
            spine.set_position(('outward', 10))  # outward by 10 points
            spine.set_smart_bounds(True)
        else:
            spine.set_color('none')  # don't draw spine

    # turn off ticks where there is no spine
    if 'left' in spines:
        ax.yaxis.set_ticks_position('left')
    else:
        # no yaxis ticks
        ax.yaxis.set_ticks([])

    if 'bottom' in spines:
        ax.xaxis.set_ticks_position('bottom')
    else:
        # no xaxis ticks
        ax.xaxis.set_ticks([])
fig = plt.figure()

x = np.linspace(0, 2*np.pi, 100)
y = 2*np.sin(x)

ax = fig.add_subplot(2, 2, 1)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['left'])

ax = fig.add_subplot(2, 2, 2)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, [])

ax = fig.add_subplot(2, 2, 3)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['left', 'bottom'])

ax = fig.add_subplot(2, 2, 4)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['bottom'])

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Plot the sparsity pattern of arrays

from matplotlib.pyplot import figure, show
import numpy

fig = figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

x = numpy.random.randn(20,20)
x[5] = 0.
x[:,12] = 0.

ax1.spy(x, markersize=5)
ax2.spy(x, precision=0.1, markersize=5)

ax3.spy(x)
ax4.spy(x, precision=0.1)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.181  pylab_examples example code: stackplot_demo.py

![Stackplot example](image)
import numpy as np
from matplotlib import pyplot as plt

fnx = lambda : np.random.randint(5, 50, 10)
y = np.row_stack((fnx(), fnx(), fnx()))
x = np.arange(10)

y1, y2, y3 = fnx(), fnx(), fnx()

fig, ax = plt.subplots()
ax.stackplot(x, y)
plt.show()

fig, ax = plt.subplots()
ax.stackplot(x, y1, y2, y3)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
def layers(n, m):
    """Return *n* random Gaussian mixtures, each of length *m*.
    """
    def bump(a):
        x = 1 / (.1 + np.random.random())
        y = 2 * np.random.random() - .5
        z = 10 / (.1 + np.random.random())
        for i in range(m):
            w = (i / float(m) - y) * z
            a[i] += x * np.exp(-w * w)
    a = np.zeros((m, n))
    for i in range(n):
        for j in range(5):
            bump(a[:, i])
    return a
d = layers(3, 100)
```python
plt.subplots()
plt.stackplot(range(100), d.T, baseline='wiggle')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.183 `pylab_examples` example code: `stem_plot.py`

```python
#!/usr/bin/env python
from pylab import *

x = linspace(0.1, 2*pi, 10)
markerline, stemlines, baseline = stem(x, cos(x), '-.')
setp(markerline, 'markerfacecolor', 'b')
setp(baseline, 'color','r', 'linewidth', 2)
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import numpy as np
from numpy import ma
from matplotlib.pyplot import step, legend, xlim, ylim, show

x = np.arange(1, 7, 0.4)
y0 = np.sin(x)
y = y0.copy() + 2.5

step(x, y, label='pre (default)')

y -= 0.5
step(x, y, where='mid', label='mid')

y -= 0.5
step(x, y, where='post', label='post')

y = ma.masked_where((y0>-0.15)&(y0<0.15), y - 0.5)
step(x, y, label='masked (pre)')

legend()
xlim(0, 7)
```
ylim(-0.5, 4)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.185 pylab_examples example code: stix_fonts_demo.py
#!/usr/bin/env python

from __future__ import unicode_literals
import os, sys, re
import gc

stests = [
    r'$\mathcircled{123} \mathrm{\mathcircled{123}} \mathbf{\mathcircled{123}}$',
    r'$\mathsf{Sans \Omega} \mathrm{\mathsf{Sans \Omega}} \mathbf{\mathsf{Sans \Omega}}$',
    r'$\mathtt{Monospace}$',
    r'$\mathcal{CALLIGRAPHIC}$',
    r'$\mathbb{Blackboard \pi}$',
    r'$\mathrm{\mathbb{Blackboard \pi}}$',
    r'$\mathfrak{Fraktur} \mathbf{\mathfrak{Fraktur}}$',
    r'$\mathscr{Script}$'
]

if sys.maxunicode > 0xffff:
    s = r'Direct Unicode: $\u23ce \mathrm{\ue0f2 \U0001D538}$'

from pylab import *

def doall():
    tests = stests

    figure(figsize=(8, (len(tests) * 1) + 2))
    plot([0, 0], 'r')
    grid(False)
    axis([0, 3, -len(tests), 0])
    yticks(arange(len(tests)) * -1)
    for i, s in enumerate(tests):
        #print (i, s.encode("ascii", "backslashreplace"))
        text(0.1, -i, s, fontsize=32)

    savefig('stix_fonts_example')
    #close('all')
    show()

if '--latex' in sys.argv:
    fd = open("stix_fonts_examples.ltx", "w")
    fd.write("\documentclass{article}\n")
    fd.write("\begin{document}\n")
    fd.write("\begin{enumerate}\n")

    for i, s in enumerate(stests):
        s = re.sub(r"(?<!\$)\$\$", "$\$", s)
        fd.write("\item %s\n" % s)

    fd.write("\end{enumerate}\n")
    fd.write("\end{document}\n")
    fd.close()
os.system("pdflatex stix_fonts_examples.ltx")
else:
doall()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.186 pylab_examples example code: stock_demo.py

```python
#!/usr/bin/env python

from matplotlib.ticker import MultipleLocator
from pylab import *
from data_helper import get_two_stock_data

d1, p1, d2, p2 = get_two_stock_data()

fig, ax = subplots()
lines = plot(d1, p1, 'bs', d2, p2, 'go')
xlabel('Days')
ylabel('Normalized price')
xlim(0, 3)
ax.xaxis.set_major_locator(MultipleLocator(1))
```

INTC vs AAPL

Normalized price

INTC vs AAPL

INTC

AAPL

Days

Normalized price

INTC

AAPL

0 1 2 3

0.75 0.80 0.85 0.90 0.95 1.00 1.05 1.10 1.15

Chapter 84. pylab_examples Examples
title('INTC vs AAPL')
legend( ('INTC', 'AAPL') )
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.187  pylab_examples example code: subplot_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)
```

"""
Simple demo with multiple subplots.
"""

84.187  pylab_examples example code: subplot_demo.py
plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'ko-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.188 pylab_examples example code: subplot_toolbar.py
Click on slider to adjust subplot param

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>0.12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bottom</td>
<td>0.10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>right</td>
<td></td>
<td>0.90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>top</td>
<td></td>
<td></td>
<td>0.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wspace</td>
<td></td>
<td></td>
<td></td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>hspace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.20</td>
</tr>
</tbody>
</table>

Reset

#!/usr/bin/env python

from pylab import *

fig = figure()
subplot(221)
imshow(rand(100,100))
subplot(222)
imshow(rand(100,100))
subplot(223)
imshow(rand(100,100))
subplot(224)
imshow(rand(100,100))

subplot_tool()
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from pylab import *

subplot(211)
imshow(rand(100,100), cmap=cm.BuPu_r)
subplot(212)
imshow(rand(100,100), cmap=cm.BuPu_r)

subplots_adjust(bottom=0.1, right=0.8, top=0.9)
cax = axes([0.85, 0.1, 0.075, 0.8])
colorbar(cax=cax)
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.190 **pylab_examples example code: subplots_demo.py**

![Simple plot](image)
Sharing both axes
Sharing x per column, y per row
Examples illustrating the use of plt.subplots().

This function creates a figure and a grid of subplots with a single call, while providing reasonable control over how the individual plots are created. For very refined tuning of subplot creation, you can still use add_subplot() directly on a new figure.

---

```python
import matplotlib.pyplot as plt
import numpy as np

# Simple data to display in various forms
x = np.linspace(0, 2 * np.pi, 400)
y = np.sin(x ** 2)

plt.close('all')

# Just a figure and one subplot
f, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Two subplots, the axes array is 1-d
f, axarr = plt.subplots(2, sharex=True)
axarr[0].plot(x, y)
```

---

84.190.  pylab_examples example code: subplots_demo.py 2303
axarr[0].set_title('Sharing X axis')
axarr[1].scatter(x, y)

# Two subplots, unpack the axes array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Three subplots sharing both x/y axes
f, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing both axes')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')

# Fine-tune figure; make subplots close to each other and hide x ticks for
# all but bottom plot.
f.subplots_adjust(hspace=0)
plt.setp([a.get_xticklabels() for a in f.axes[:-1]], visible=False)

# row and column sharing
f, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex='col', sharey='row')
ax1.plot(x, y)
ax1.set_title('Sharing x per column, y per row')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')
ax4.plot(x, 2 * y ** 2 - 1, color='r')

# Four axes, returned as a 2-d array
f, axarr = plt.subplots(2, 2)
axarr[0, 0].plot(x, y)
axarr[0, 0].set_title('Axis [0,0]')
axarr[0, 1].scatter(x, y)
axarr[0, 1].set_title('Axis [0,1]')
axarr[1, 0].plot(x, y ** 2)
axarr[1, 0].set_title('Axis [1,0]')
axarr[1, 1].scatter(x, y ** 2)
axarr[1, 1].set_title('Axis [1,1]')

# Fine-tune figure; hide x ticks for top plots and y ticks for right plots
plt.setp([a.get_xticklabels() for a in axarr[0, :]], visible=False)
plt.setp([a.get_yticklabels() for a in axarr[:, 1]], visible=False)

# Four polar axes
plt.subplots(2, 2, subplot_kw=dict(polar=True))

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python
from pylab import *

dt = 0.01
x = arange(-50.0, 50.0, dt)
y = arange(0, 100.0, dt)

subplot(311)
plot(x, y)
xscale('symlog')
ylabel('symlogx')
gca().yaxis.grid(True)
grid(True)
# minor grid on too

subplot(312)
plot(y, x)
ylabel('symlogy')

subplot(313)
plot(x, np.sin(x / 3.0))
Matplotlib, Release 1.4.3

```python
import time
from pylab import *

def get_memory():
    "Simulate a function that returns system memory"
    return 100*(0.5+0.5*sin(0.5*pi*time.time()))

def get_cpu():
    "Simulate a function that returns cpu usage"
    return 100*(0.5+0.5*sin(0.2*pi*(time.time()-0.25)))

def get_net():
    "Simulate a function that returns network bandwidth"
    return 100*(0.5+0.5*sin(0.7*pi*(time.time()-0.1)))

def get_stats():
    return get_memory(), get_cpu(), get_net()

fig, ax = plt.subplots()
ind = arange(1,4)
pm, pc, pn = bar(ind, get_stats())

centers = ind + 0.5*pm.get_width()
pm.set_facecolor('r')
pc.set_facecolor('g')
pn.set_facecolor('b')

ax.set_xlim([0.5,4])
ax.set_xticks(centers)
ax.set_ylim([0,100])
ax.set_xticklabels(['Memory', 'CPU', 'Bandwidth'])
ax.set_ylabel('Percent usage')
ax.set_title('System Monitor')
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
for i in range(200):  # run for a little while
    m, c, n = get_stats()
    pm.set_height(m)
    pc.set_height(c)
    pn.set_height(n)
    ax.set_ylim([0, 100])
    draw()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.193 pylab_examples example code: table_demo.py

Demo of table function to display a table within a plot.

import numpy as np
import matplotlib.pyplot as plt

data = [[66386, 174296, 75131, 577908, 32015],
        [431.5, 1049.4, 799.6, 2149.8, 917.9],
        [292.2, 717.8, 456.4, 1368.5, 865.6],
        [213.8, 636.0, 305.7, 1175.2, 796.0],
        [124.6, 555.4, 153.2, 677.2, 192.5],
        [66.4, 174.3, 75.1, 577.9, 32.0],
        [0, 500, 1000, 1500, 2000],
        ['Freeze', 'Wind', 'Flood', 'Quake', 'Hail'],
        ['100 year', '431.5', '1049.4', '799.6', '2149.8', '917.9'],
        ['50 year', '292.2', '717.8', '456.4', '1368.5', '865.6'],
        ['20 year', '213.8', '636.0', '305.7', '1175.2', '796.0'],
        ['10 year', '124.6', '555.4', '153.2', '677.2', '192.5'],
        ['5 year', '66.4', '174.3', '75.1', '577.9', '32.0']]

84.193. pylab_examples example code: table_demo.py 2307
columns = ("Freeze", 'Wind', 'Flood', 'Quake', 'Hail')
rows = ["%d year" % x for x in (100, 50, 20, 10, 5)]

values = np.arange(0, 2500, 500)
value_increment = 1000

# Get some pastel shades for the colors
colors = plt.cm.BuPu(np.linspace(0, 0.5, len(columns)))

n_rows = len(data)
index = np.arange(len(columns)) + 0.3
bar_width = 0.4

# Initialize the vertical-offset for the stacked bar chart.
y_offset = np.array([0.0] * len(columns))

# Plot bars and create text labels for the table
cell_text = []
for row in range(n_rows):
    plt.bar(index, data[row], bar_width, bottom=y_offset, color=colors[row])
    y_offset = y_offset + data[row]
    cell_text.append(['%1.1f' % (x/1000.0) for x in y_offset])
# Reverse colors and text labels to display the last value at the top.
colors = colors[::-1]
cell_text.reverse()

# Add a table at the bottom of the axes
the_table = plt.table(cellText=cell_text,
                       rowLabels=rows,
                       rowColours=colors,
                       colLabels=columns,
                       loc='bottom')

# Adjust layout to make room for the table:
plt.subplots_adjust(left=0.2, bottom=0.2)

plt.ylabel("Loss in \$\{0}'s".format(value_increment))
plt.yticks(values * value_increment, ["%d" % val for val in values])
plt.xticks([])
plt.title('Loss by Disaster')
plt.show()
You can use TeX to render all of your matplotlib text if the rc parameter text.usetex is set. This works currently on the agg and ps backends, and requires that you have tex and the other dependencies described at http://matplotlib.org/users/usetex.html properly installed on your system. The first time you run a script you will see a lot of output from tex and associated tools. The next time, the run may be silent, as a lot of the information is cached in ~/.tex.cache

\begin{verbatim}
import numpy as np
import matplotlib.pyplot as plt

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.figure(1, figsize=(6,4))
tax = plt.axes([0.1, 0.1, 0.8, 0.7])
t = np.linspace(0.0, 1.0, 100)
s = np.cos(4 * np.pi * t) + 2
plt.plot(t, s)
plt.xlabel(r'$\textbf{time (s)}$')
plt.ylabel(r'$\textit{voltage (mV)}$', fontsize=16)
\end{verbatim}
mathtext example with unicode

\[ \sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}! \]

```
# This demo is tex_demo.py modified to have unicode. See that file for more information.

from __future__ import unicode_literals
import matplotlib as mpl
mpl.rcParams['text.usetex']=True
mpl.rcParams['text.latex.unicode']=True
from numpy import arange, cos, pi
from matplotlib.pyplot import figure, axes, plot, xlabel, ylabel, title, grid, savefig, show

figure(1, figsize=(6,4))
```
```python
ax = axes([0.1, 0.1, 0.8, 0.7])
t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*pi*t)+2
plot(t, s)

xlabel(r'	extbf{time (s)}')
ylabel(r'	extit{Velocity (°/sec)}', fontsize=16)
title(r'\TeX\ is Number \(\sum_{n=1}^{\infty}\frac{-e^{i\pi}}{2^n}\)',
      fontsize=16, color='r')
grid(True)
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.196  pylab_examples example code: text_handles.py

```python
#!/usr/bin/env python
#Controlling the properties of axis text using handles

# See examples/text_themes.py for a more elegant, pythonic way to control
# fonts. After all, if we were slaves to MATLAB, we wouldn't be
# using python!
```
from pylab import *

def f(t):
    s1 = sin(2*pi*t)
    e1 = exp(-t)
    return multiply(s1, e1)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)

fig, ax = plt.subplots()
plot(t1, f(t1), 'bo', t2, f(t2), 'k')
text(3.0, 0.6, 'f(t) = exp(-t) sin(2 pi t)')
ttext = title('Fun with text!')
ytext = ylabel('Damped oscillation')
xtext = xlabel('time (s)')
setp(ttext, size='large', color='r', style='italic')
setp(xtext, size='medium', name='courier', weight='bold', color='g')
setp(ytext, size='medium', name='helvetica', weight='light', color='b')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
The way matplotlib does text layout is counter-intuitive to some, so this example is designed to make it a little clearer. The text is aligned by it's bounding box (the rectangular box that surrounds the ink rectangle). The order of operations is basically rotation then alignment, rather than alignment then rotation. Basically, the text is centered at your x,y location, rotated around this point, and then aligned according to the bounding box of the rotated text.

So if you specify left, bottom alignment, the bottom left of the bounding box of the rotated text will be at the x,y coordinate of the text.

But a picture is worth a thousand words!
"""

from pylab import *

def addtext(props):
    text(0.5, 0.5, 'text 0', props, rotation=0)
    text(1.5, 0.5, 'text 45', props, rotation=45)
    text(2.5, 0.5, 'text 135', props, rotation=135)
    text(3.5, 0.5, 'text 225', props, rotation=225)
    text(4.5, 0.5, 'text -45', props, rotation=-45)
text(3.5, 0.5, 'text 225', props, rotation=225)

grid(True)

# the text bounding box
bbox = {'fc':0.8, 'pad':0}

subplot(211)
addtext({'ha':'center', 'va':'center', 'bbox':bbox})
xlim(0,5)
xticks(arange(0, 5.1, 0.5), [])
ylabel('center / center')

subplot(212)
addtext({'ha':'left', 'va':'bottom', 'bbox':bbox})
xlim(0,5)
xticks(arange(0, 5.1, 0.5))
ylabel('left / bottom')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.198 pylab_examples example code: text_rotation_relative_to_line.py
```python
#!/usr/bin/env python

""
Text objects in matplotlib are normally rotated with respect to the screen coordinate system (i.e., 45 degrees rotation plots text along a line that is in between horizontal and vertical no matter how the axes are changed). However, at times one wants to rotate text with respect to something on the plot. In this case, the correct angle won't be the angle of that object in the plot coordinate system, but the angle that that object APPEARS in the screen coordinate system. This angle is found by transforming the angle from the plot to the screen coordinate system, as shown in the example below.
""

from pylab import *

# Plot diagonal line (45 degrees)
h = plot( r_[:10], r_[:10] )

# set limits so that it no longer looks on screen to be 45 degrees
xlim([-10,20])

# Locations to plot text
l1 = array((1,1))
l2 = array((5,5))

# Rotate angle
angle = 45
trans_angle = gca().transData.transform_angles(array((45,)),
                                             l2.reshape((1,2)))[0]

# Plot text
th1 = text(l1[0],l1[1],'text not rotated correctly',fontsize=16,
            rotation=angle)
th2 = text(l2[0],l2[1],'text not rotated correctly',fontsize=16,
            rotation=trans_angle)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
Matplotlib can display plot titles centered, flush with the left side of a set of axes, and flush with the right side of a set of axes.

```
#!/usr/bin/env python

""
matplotlib can display plot titles centered, flush with the left side of a set of axes, and flush with the right side of a set of axes.
""

import matplotlib.pyplot as plt

plt.plot(range(10))

plt.title('Center Title')
plt.title('Left Title', loc='left')
plt.title('Right Title', loc='right')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
Use backend agg to access the figure canvas as an RGB string and then convert it to an array and pass it to Pillow for rendering.
"""

import pylab
from matplotlib.backends.backend_agg import FigureCanvasAgg

try:
    from PIL import Image
except ImportError:
    raise SystemExit("PIL must be installed to run this example")

pylab.plot([1,2,3])

canvas = pylab.get_current_fig_manager().canvas
agg = canvas.switch_backends(FigureCanvasAgg)
agg.draw()

s = agg.tostring_rgb()
# get the width and the height to resize the matrix
l,b,w,h = agg.figure.bbox.bounds
w, h = int(w), int(h)

X = pylab.fromstring(s, pylab.uint8)
X.shape = h, w, 3

im = Image.fromstring("RGB", (w,h), s)

# Uncomment this line to display the image using ImageMagick's
# `display` tool.
# im.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.201 pylab_examples example code: toggle_images.py

#!/usr/bin/env python

""" toggle between two images by pressing "t"

The basic idea is to load two images (they can be different shapes) and plot
them to the same axes with hold "on". Then, toggle the visible property of them using keypress event handling

If you want two images with different shapes to be plotted with the same extent, they must have the same "extent" property

As usual, we'll define some random images for demo. Real data is much more exciting!

Note, on the wx backend on some platforms (e.g., linux), you have to first click on the figure before the keypress events are activated. If you know how to fix this, please email us!

```python
from pylab import *

# two images x1 is initially visible, x2 is not
x1 = rand(100, 100)
x2 = rand(150, 175)

# arbitrary extent - both images must have same extent if you want
# them to be resampled into the same axes space
extent = (0,1,0,1)
im1 = imshow(x1, extent=extent)
im2 = imshow(x2, extent=extent, hold=True)
im2.set_visible(False)

def toggle_images(event):
    'toggle the visible state of the two images'
    if event.key != 't': return
    b1 = im1.get_visible()
b2 = im2.get_visible()
im1.set_visible(not b1)
im2.set_visible(not b2)
draw()

connect('key_press_event', toggle_images)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
84.202  pylab_examples example code: transoffset.py
#!/usr/bin/env python

This illustrates the use of transforms.offset_copy to make a transform that positions a drawing element such as a text string at a specified offset in screen coordinates (dots or inches) relative to a location given in any coordinates.

Every Artist--the mpl class from which classes such as Text and Line are derived--has a transform that can be set when the Artist is created, such as by the corresponding pylab command. By default this is usually the Axes.transData transform, going from data units to screen dots. We can use the offset_copy function to make a modified copy of this transform, where the modification consists of an offset.

import pylab as P
from matplotlib.transforms import offset_copy

X = P.arange(7)
Y = X**2

fig = P.figure(figsize=(5,10))
ax = P.subplot(2,1,1)

# If we want the same offset for each text instance,
# we only need to make one transform. To get the
# transform argument to offset_copy, we need to make the axes
# first; the subplot command above is one way to do this.

transOffset = offset_copy(ax.transData, fig=fig, x = 0.05, y=0.10, units='inches')

for x, y in zip(X, Y):
    P.plot((x,),(y,), 'ro')
    P.text(x, y, '
%d
%d
' % (int(x),int(y)), transform=transOffset)

# offset_copy works for polar plots also.

ax = P.subplot(2,1,2, polar=True)

transOffset = offset_copy(ax.transData, fig=fig, y = 6, units='dots')

for x, y in zip(X, Y):
    P.polar((x,),(y,), 'ro')
    P.text(x, y, '
%d
%d
' % (int(x),int(y)),
        transform=transOffset,
        horizontalalignment='center',
        verticalalignment='bottom')

2322 Chapter 84. pylab_examples Examples
P.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.203 pylab_examples example code: tricontour_demo.py

Contour plot of Delaunay triangulation
Contour plots of unstructured triangular grids.

import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
n_angles = 48
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()
# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# pcolor plot.
plt.figure()
plt.gca().set_aspect('equal')
plt.tricontourf(triang, z)
plt.colorbar()
plt.tricontour(triang, z, colors='k')
plt.title('Contour plot of Delaunay triangulation')

# You can specify your own triangulation rather than perform a Delaunay
# triangulation of the points, where each triangle is given by the indices of
# the three points that make up the triangle, ordered in either a clockwise or
# anticlockwise manner.

xy = np.asarray(
    [[-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
     [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
     [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.089, 0.919],
     [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
     [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
     [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
     [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
     [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
     [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
     [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
     [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
     [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
     [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
     [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
     [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
     [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
     [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
     [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
     [-0.077, 0.996], [-0.059, 0.993]])
x = xy[:, 0]*180/3.14159
y = xy[:, 1]*180/3.14159
x0 = -5
y0 = 52
z = np.exp(-0.01*((x-x0)*(x-x0) + (y-y0)*(y-y0)))
triangles = np.asarray(
    [[67, 66, 1], [65, 2, 66], [1, 66, 2], [64, 2, 65], [63, 3, 64],
     [60, 59, 57], [2, 64, 3], [3, 63, 4], [0, 67, 1], [62, 4, 63],
     [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [4, 62, 68],
     [6, 5, 9], [61, 68, 62], [69, 68, 61], [9, 5, 70], [6, 8, 7],
)
# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to triplot directly. It would be better to use a Triangulation
# object if the same triangulation was to be used more than once to save
# duplicated calculations.
plt.figure()
plt.gca().set_aspect('equal')
plt.tricontourf(x, y, triangles, z)
plt.colorbar()
plt.title('Contour plot of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Filtering a Delaunay mesh
(application to high-resolution tricontouring)

Demonstrates high-resolution tricontouring of a random set of points; a matplotlib.tri.TriAnalyzer is used to improve the plot quality.

The initial data points and triangular grid for this demo are:
- a set of random points is instantiated, inside [-1, 1] x [-1, 1] square
- A Delaunay triangulation of these points is then computed, of which a random subset of triangles is masked out by the user (based on *init_mask_frac* parameter). This simulates invalidated data.

The proposed generic procedure to obtain a high resolution contouring of such a data set is the following:
1) Compute an extended mask with a matplotlib.tri.TriAnalyzer, which will exclude badly shaped (flat) triangles from the border of the triangulation. Apply the mask to the triangulation (using set_mask).
2) Refine and interpolate the data using a matplotlib.tri.UniformTriRefiner.
3) Plot the refined data with tricontour.

```python
from matplotlib.tri import Triangulation, TriAnalyzer, UniformTriRefiner
import matplotlib.pyplot as plt
```

...
import matplotlib.cm as cm
import numpy as np

# Analytical test function
#-------------------------------------------------------------
def experiment_res(x, y):
    """ An analytic function representing experiment results """
    x = 2.*x
    r1 = np.sqrt((0.5-x)**2 + (0.5-y)**2)
    thetal1 = np.arctan2(0.5-x, 0.5-y)
    r2 = np.sqrt((-x-0.2)**2 + (-y-0.2)**2)
    thetal2 = np.arctan2(-x-0.2, -y-0.2)
    z = (4*(np.exp((r1/10)**2)-1)*30. * np.cos(3*thetal1) +
         (np.exp((r2/10)**2)-1)*30. * np.cos(5*thetal2) +
         2*(x**2 + y**2))
    return (np.max(z)-z)/(np.max(z)-np.min(z))

# Generating the initial data test points and triangulation for the demo
#-------------------------------------------------------------
# User parameters for data test points
n_test = 200 # Number of test data points, tested from 3 to 5000 for subdiv=3

subdiv = 3 # Number of recursive subdivisions of the initial mesh for smooth # plots. Values >3 might result in a very high number of triangles # for the refine mesh: new triangles numbering = (4**subdiv)*ntri

init_mask_frac = 0.0 # Float > 0. adjusting the proportion of # (invalid) initial triangles which will be masked # out. Enter 0 for no mask.

min_circle_ratio = .01 # Minimum circle ratio - border triangles with circle # ratio below this will be masked if they touch a # border. Suggested value 0.01 ; Use -1 to keep # all triangles.

# Random points
random_gen = np.random.mtrand.RandomState(seed=127260)
x_test = random_gen.uniform(-1., 1., size=n_test)
y_test = random_gen.uniform(-1., 1., size=n_test)
z_test = experiment_res(x_test, y_test)

# meshing with Delaunay triangulation
tri = Triangulation(x_test, y_test)
ntri = tri.triangles.shape[0]

# Some invalid data are masked out
mask_init = np.zeros(ntri, dtype=np.bool)
masked_tri = random_gen.randint(0, ntri, int(ntri*init_mask_frac))
mask_init[masked_tri] = True
tri.set_mask(mask_init)
# Improving the triangulation before high-res plots: removing flat triangles
# masking badly shaped triangles at the border of the triangular mesh.
mask = TriAnalyzer(tri).get_flat_tri_mask(min_circle_ratio)
tri.set_mask(mask)

# refining the data
refiner = UniformTriRefiner(tri)
tri_refi, z_test_refi = refiner.refine_field(z_test, subdiv=subdiv)

# analytical 'results' for comparison
z_expected = experiment_res(tri_refi.x, tri_refi.y)

# for the demo: loading the 'flat' triangles for plot
flat_tri = Triangulation(x_test, y_test)
flat_tri.set_mask(~mask)

# Now the plots
# User options for plots
plot_tri = True  # plot of base triangulation
plot_masked_tri = True  # plot of excessively flat excluded triangles
plot_refi_tri = False  # plot of refined triangulation
plot_expected = False  # plot of analytical function values for comparison

# Graphical options for tricontouring
levels = np.arange(0., 1., 0.025)
cmap = cm.get_cmap(name='Blues', lut=None)

plt.figure()
plt.gca().set_aspect('equal')
plt.title("Filtering a Delaunay mesh\n" + "(application to high-resolution tricontouring)")

# 1) plot of the refined (computed) data countours:
plt.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
linewidens=[2.0, 0.5, 1.0, 0.5])
# 2) plot of the expected (analytical) data countours (dashed):
if plot_expected:
    plt.tricontour(tri_refi, z_expected, levels=levels, cmap=cmap,
linestyles='--')
# 3) plot of the fine mesh on which interpolation was done:
if plot_refi_tri:
    plt.triplot(tri_refi, color='0.97')
# 4) plot of the initial 'coarse' mesh:
if plot_tri:
    plt.triplot(tri, color='0.7')
# 4) plot of the unvalidated triangles from naive Delaunay Triangulation:
if plot_masked_tri:
    plt.triplot(flat_tri, color='red')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.205 pylab_examples example code: tricontour_smooth_user.py

Demonstrates high-resolution tricontouring on user-defined triangular grids with matplotlib.tri.UniformTriRefiner

from matplotlib.tri import Triangulation, UniformTriRefiner
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math

# Analytical test function
```python
# A function of 2 variables
r1 = np.sqrt((0.5-x)**2 + (0.5-y)**2)
theta1 = np.arctan2(0.5-x, 0.5-y)
r2 = np.sqrt((-x-0.2)**2 + (-y-0.2)**2)
theta2 = np.arctan2(-x-0.2, -y-0.2)
z = -(2*(np.exp((r1/10)**2)-1)*30. * np.cos(7.*theta1) + 
(nexp(r2/10)**2-1)*30. * np.cos(11.*theta2) + 
0.7*(x**2 + y**2))
return (np.max(z)-z)/(np.max(z)-np.min(z))

# Creating a Triangulation

# First create the x and y coordinates of the points.
angles = 20
n_radii = 10
min_radius = 0.15
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = function_z(x, y)

# Now create the Triangulation.
# (Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.)
triang = Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# Refine data

refiner = UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(z, subdiv=3)

# Plot the triangulation and the high-res iso-contours
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, lw=0.5, color='white')
```

84.205. `pylab_examples` example code: `tricontour_smooth_user.py` 2331
```python
levels = np.arange(0., 1., 0.025)
cmap = cm.get_cmap(name='terrain', lut=None)
plt.tricontourf(tri_refi, z_test_refi, levels=levels, cmap=cmap)
plt.tricontour(tri_refi, z_test_refi, levels=levels,
               colors=['0.25', '0.5', '0.5', '0.5', '0.5'],
               linewidths=[1.0, 0.5, 0.5, 0.5, 0.5])
plt.title("High-resolution tricontouring")
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.206 `pylab_examples` example code: `tricontour_vs_griddata.py`

""
Comparison of griddata and tricontour for an unstructured triangular grid.
""

```python
from __future__ import print_function
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
```
from numpy.random import uniform, seed
from matplotlib.mlab import griddata
import time
seed(0)
npts = 200
ngridx = 100
ngridy = 200
x = uniform(-2, 2, npts)
y = uniform(-2, 2, npts)
z = x**np.exp(-x**2 - y**2)

# griddata and contour.
start = time.clock()
plt.subplot(211)
xi = np.linspace(-2.1, 2.1, ngridx)
iy = np.linspace(-2.1, 2.1, ngridy)
zi = griddata(x, y, z, xi, iy, interp='linear')
plt.contour(xi, iy, zi, 15, linewidths=0.5, colors='k')
plt.contourf(xi, iy, zi, 15, cmap=plt.cm.rainbow,
             norm=plt.Normalize(vmax=abs(zi).max(), vmin=-abs(zi).max()))
plt.colorbar()
plt.plot(x, y, 'ko', ms=3)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('griddata and contour (%d points, %d grid points)
          % (npts, ngridx*ngridy))
print ('griddata and contour seconds: %f' % (time.clock() - start))

# tricontour.
start = time.clock()
plt.subplot(212)
triang = tri.Triangulation(x, y)
plt.tricontour(x, y, z, 15, linwidths=0.5, colors='k')
plt.tricontourf(x, y, z, 15, cmap=plt.cm.rainbow,
                norm=plt.Normalize(vmax=abs(zi).max(), vmin=-abs(zi).max()))
plt.colorbar()
plt.plot(x, y, 'ko', ms=3)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('tricontour (%d points) % npts)
print ('tricontour seconds: %f' % (time.clock() - start))
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demonstrates computation of gradient with matplotlib.tri.CubicTriInterpolator.

```python
from matplotlib.tri import Triangulation, UniformTriRefiner,
    CubicTriInterpolator
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math

# Electrical potential of a dipole

def dipole_potential(x, y):
    """The electric dipole potential V""
    r_sq = x**2 + y**2
    theta = np.arctan2(y, x)
    z = np.cos(theta)/r_sq
    return (np.max(z)-z) / (np.max(z)-np.min(z))
```

Gradient plot: an electrical dipole
# Creating a Triangulation

# First create the x and y coordinates of the points.

n_angles = 30
n_radii = 10
min_radius = 0.2
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
V = dipole_potential(x, y)

# Create the Triangulation; no triangles specified so Delaunay triangulation
# created.
triang = Triangulation(x, y)

# Mask off unwanted triangles.

xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# Refine data - interpolates the electrical potential V

refiner = UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(V, subdiv=3)

# Computes the electrical field (Ex, Ey) as gradient of electrical potential

tci = CubicTriInterpolator(triang, -V)
# Gradient requested here at the mesh nodes but could be anywhere else:
(Ex, Ey) = tci.gradient(triang.x, triang.y)
E_norm = np.sqrt(Ex**2 + Ey**2)

# Plot the triangulation, the potential iso-contours and the vector field

plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, color='0.8')
levels = np.arange(0., 1., 0.01)
cmap = cm.get_cmap(name='hot', lut=None)
plt.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
            linewidths=[2.0, 1.0, 1.0, 1.0])
# Plots direction of the electrical vector field
plt.quiver(triang.x, triang.y, Ex/E_norm, Ey/E_norm,
    units='xy', scale=10., zorder=3, color='blue',
    width=0.007, headwidth=3., headlength=4.)

plt.title('Gradient plot: an electrical dipole')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.208 pylab_examples example code: triinterp_demo.py

```python
""
Interpolation from triangular grid to quad grid.
""
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
import numpy as np

# Create triangulation.
x = np.asarray([0, 1, 2, 3, 0.5, 1.5, 2.5, 1, 2, 1.5])
y = np.asarray([0, 0, 0, 0, 1.0, 1.0, 1.0, 2, 2, 3.0])
triangles = [[0, 1, 4], [1, 2, 5], [2, 3, 6], [1, 5, 4], [2, 6, 5], [4, 5, 7],
```

---

84.208 *pylab_examples* example code: triinterp_demo.py

```python
""
Interpolation from triangular grid to quad grid.
""
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
import numpy as np

# Create triangulation.
x = np.asarray([0, 1, 2, 3, 0.5, 1.5, 2.5, 1, 2, 1.5])
y = np.asarray([0, 0, 0, 0, 1.0, 1.0, 1.0, 2, 2, 3.0])
triangles = [[0, 1, 4], [1, 2, 5], [2, 3, 6], [1, 5, 4], [2, 6, 5], [4, 5, 7],
```

---

84.208 *pylab_examples* example code: triinterp_demo.py

```python
""
Interpolation from triangular grid to quad grid.
""
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
import numpy as np

# Create triangulation.
x = np.asarray([0, 1, 2, 3, 0.5, 1.5, 2.5, 1, 2, 1.5])
y = np.asarray([0, 0, 0, 0, 1.0, 1.0, 1.0, 2, 2, 3.0])
triangles = [[0, 1, 4], [1, 2, 5], [2, 3, 6], [1, 5, 4], [2, 6, 5], [4, 5, 7],
```

---

84.208 *pylab_examples* example code: triinterp_demo.py

```python
""
Interpolation from triangular grid to quad grid.
""
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
import numpy as np

# Create triangulation.
x = np.asarray([0, 1, 2, 3, 0.5, 1.5, 2.5, 1, 2, 1.5])
y = np.asarray([0, 0, 0, 0, 1.0, 1.0, 1.0, 2, 2, 3.0])
triangles = [[0, 1, 4], [1, 2, 5], [2, 3, 6], [1, 5, 4], [2, 6, 5], [4, 5, 7],
```
```python
[5, 6, 8], [5, 8, 7], [7, 8, 9]]
triang = mtri.Triangulation(x, y, triangles)

# Interpolate to regularly-spaced quad grid.
z = np.cos(1.5*x)*np.cos(1.5*y)
xi, yi = np.meshgrid(np.linspace(0, 3, 20), np.linspace(0, 3, 20))

interp_lin = mtri.LinearTriInterpolator(triang, z)
zi_lin = interp_lin(xi, yi)

interp_cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
zi_cubic_geom = interp_cubic_geom(xi, yi)

interp_cubic_min_E = mtri.CubicTriInterpolator(triang, z, kind='min_E')
zi_cubic_min_E = interp_cubic_min_E(xi, yi)

# Plot the triangulation.
plt.subplot(221)
plt.tricontourf(triang, z)
plt.triplot(triang, 'ko-')
plt.title('Triangular grid')

# Plot linear interpolation to quad grid.
plt.subplot(222)
plt.contourf(xi, yi, zi_lin)
plt.plot(xi, yi, 'k-', alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', alpha=0.5)
plt.title('Linear interpolation')

# Plot cubic interpolation to quad grid, kind=geom
plt.subplot(223)
plt.contourf(xi, yi, zi_cubic_geom)
plt.plot(xi, yi, 'k-', alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', alpha=0.5)
plt.title('Cubic interpolation, kind=geom')

# Plot cubic interpolation to quad grid, kind=min_E
plt.subplot(224)
plt.contourf(xi, yi, zi_cubic_min_E)
plt.plot(xi, yi, 'k-', alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', alpha=0.5)
plt.title('Cubic interpolation, kind=min_E')

plt.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
tripcolor of Delaunay triangulation, flat shading
pcolor of Delaunay triangulation, gouraud shading
Pseudocolor plots of unstructured triangular grids.

```
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()
```

---

Chapter 84. pylab_examples Examples
# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# tripcolor plot.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(triang, z, shading='flat', cmap=plt.cm.rainbow)
plt.colorbar()
plt.title('tripcolor of Delaunay triangulation, flat shading')

# Illustrate Gouraud shading.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(triang, z, shading='gouraud', cmap=plt.cm.rainbow)
plt.colorbar()
plt.title('tripcolor of Delaunay triangulation, gouraud shading')

# You can specify your own triangulation rather than perform a Delaunay
# triangulation of the points, where each triangle is given by the indices of
# the three points that make up the triangle, ordered in either a clockwise or
# anticlockwise manner.

xy = np.asarray([
[-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
[-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
[-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
[-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
[-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
[-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
[-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
[-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
[-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
[-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
[-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
[-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
[-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
[ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
[ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
[ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
[-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
[-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
[-0.077, 0.990], [-0.059, 0.993]])
x = xy[:, 0]*180/3.14159
y = xy[:, 1]*180/3.14159

triangles = np.asarray([}
xmid = x[triangles].mean(axis=1)
ymid = y[triangles].mean(axis=1)
x0 = -5
y0 = 52
zfaces = np.exp(-0.01*((xmid-x0)*(xmid-x0) + (ymid-y0)*(ymid-y0)))

# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to tripcolor directly. It would be better to use a Triangulation
# object if the same triangulation was to be used more than once to save
# duplicated calculations.
# Can specify one color value per face rather than one per point by using the
# facecolors kwarg.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(x, y, triangles, facecolors=zfaces, edgecolors='k')
plt.colorbar()
plt.title('tripcolor of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')
plt.show()
triplot of Delaunay triangulation
Creating and plotting unstructured triangular grids.

```python
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
```

triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# Plot the triangulation.
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, 'bo-')
plt.title('triplot of Delaunay triangulation')

# You can specify your own triangulation rather than perform a Delaunay
# triangulation of the points, where each triangle is given by the indices of
# the three points that make up the triangle, ordered in either a clockwise or
# anticlockwise manner.

xy = np.asarray([[-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
                 [-0.045, 0.897], [-0.037, 0.895], [-0.027, 0.900], [-0.017, 0.907],
                 [-0.009, 0.904], [-0.001, 0.907], [-0.069, 0.921], [-0.054, 0.919],
                 [-0.048, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
                 [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
                 [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
                 [-0.097, 0.975], [-0.101, 0.984], [-0.101, 0.980], [-0.108, 0.980],
                 [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
                 [-0.105, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
                 [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
                 [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
                 [-0.045, 0.980], [-0.052, 0.975], [-0.049, 0.973], [-0.026, 0.968],
                 [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
                 [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
                 [ 0.017, 0.906], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
                 [ 0.001, 0.883], [ 0.012, 0.884], [ 0.029, 0.883], [ 0.038, 0.879],
                 [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
                 [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
                 [-0.077, 0.990], [-0.059, 0.993]])
x = xy[:, 0]*180/3.14159
y = xy[:, 1]*180/3.14159

triangles = np.asarray([ [67, 66, 1], [65, 2, 66], [1, 66, 2], [64, 2, 65], [63, 3, 64],
                         [60, 59, 57], [2, 64, 3], [3, 63, 4], [0, 67, 1], [62, 4, 63],
                         [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [4, 62, 68],
                         [6, 5, 9], [61, 68, 62], [69, 68, 61], [9, 5, 70], [6, 8, 7],
                         [4, 70, 5], [8, 6, 9], [56, 69, 57], [69, 56, 52], [70, 10, 9],
                         [54, 53, 55], [56, 55, 53], [68, 70, 4], [52, 56, 53], [11, 4, 10],
                         [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
                         [52, 71, 69], [12, 10, 11], [71, 52, 50], [71, 14, 13], [50, 49, 71],
                         [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
                         [49, 51, 52], [52, 51, 50], [14, 71, 48], [17, 19, 18], [17, 20, 19],
                         [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],]}
# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to triplot directly. It would be better to use a Triangulation object
# if the same triangulation was to be used more than once to save duplicated
# calculations.
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(x, y, triangles, 'go-')
plt.title('triplot of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.axes as maxes

from matplotlib import rcParams
rcParams['text.usetex']=True
rcParams['text.latex.unicode']=True

class Axes(maxes.Axes):
    """
A hackish way to simultaneously draw texts w/ usetex=True and usetex=False in the same figure. It does not work in the ps backend.

```python
def __init__(self, *kl, **kw):
    self.usetex = kw.pop("usetex", "False")
    self.preview = kw.pop("preview", "False")

    maxes.Axes.__init__(self, *kl, **kw)

def draw(self, renderer):
    usetex = plt.rcParams["text.usetex"]
    preview = plt.rcParams["text.latex.preview"]
    plt.rcParams["text.usetex"] = self.usetex
    plt.rcParams["text.latex.preview"] = self.preview

    maxes.Axes.draw(self, renderer)

    plt.rcParams["text.usetex"] = usetex
    plt.rcParams["text.latex.preview"] = preview

Subplot = maxes.subplot_class_factory(Axes)

def test_window_extent(ax, usetex, preview):
    va = "baseline"
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

    #t = ax.text(0., 0., r"mlp", va="baseline", size=150)
    text_kw = dict(va=va,
                    size=50,
                    bbox=dict(pad=0., ec="k", fc="none"))

    test_strings = ["lg", r"\frac{1}{2}\pi",
                    r"p^{3^A}", r"p_{3_2}"
                   ]

    ax.axvline(0, color="r")

    for i, s in enumerate(test_strings):
        ax.axhline(i, color="r")
        ax.text(0., 3-i, s, **text_kw)

    ax.set_xlim(-0.1,1.1)
    ax.set_ylim(-.8,3.9)

    ax.set_title("usetex=%s\npreview=%s" % (str(usetex), str(preview)))
```

Subplot = maxes.subplot_class_factory(Axes)

```
```python
F = plt.figure(figsize=(2.*3,6.5))

for i, usetex, preview in [[0, False, False],
                           [1, True, False],
                           [2, True, True]]:
    ax = Subplot(F, 1, 3, i+1, usetex=usetex, preview=preview)
    F.add_subplot(ax)
    F.subplots_adjust(top=0.85)

test_window_extent(ax, usetex=usetex, preview=preview)

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.212  *pylab_examples* example code: *usetex_demo.py*

```python
import matplotlib
matplotlib.rc('text', usetex = True)
import pylab
```
```python
import numpy as np

## interface tracking profiles
N = 500
delta = 0.6
X = -1 + 2. * np.arange(N) / (N - 1)
pylab.plot(X, (1 - np.tanh(4. * X / delta)) / 2, 
## phase field tanh profiles
X, (X + 1) / 2,
## level set distance function
X, (1.4 + np.tanh(4. * X / delta)) / 4,
## composition profile
X, X < 0, 'k--',
## sharp interface
linewidth = 5)

## legend
pylab.legend((r'phase field', r'level set', r'composition', r'sharp interface'), shadow = True, loc = (0.01, 0.55))
ltext = pylab.gca().get_legend().get_texts()
pylab.setp(ltext[0], fontsize = 20, color = 'b')
pylab.setp(ltext[1], fontsize = 20, color = 'g')
pylab.setp(ltext[2], fontsize = 20, color = 'r')
pylab.setp(ltext[3], fontsize = 20, color = 'k')

## the arrow
height = 0.1
offset = 0.02
pylab.plot((-delta / 2., delta / 2), (height, height), 'k', linewidth = 2)
pylab.plot((-delta / 2, -delta / 2 + offset * 2), (height, height - offset), 'k', linewidth = 2)
pylab.plot((-delta / 2, -delta / 2 + offset * 2), (height, height + offset), 'k', linewidth = 2)
pylab.plot((delta / 2, delta / 2 - offset * 2), (height, height - offset), 'k', linewidth = 2)
pylab.plot((delta / 2, delta / 2 - offset * 2), (height, height + offset), 'k', linewidth = 2)
pylab.text(-0.06, height - 0.06, r'$\delta$', {color: 'k', fontsize: 24})

## X-axis label
pylab.xticks((-1, 0, 1), ('-1', '0', '1'), color = 'k', size = 20)

## Left Y-axis labels
pylab.ylabel(r'\bf{phase field} $\phi$', {color: 'b', 'fontsize': 20 })
pylab.yticks((0, 0.5, 1), ('0', '.5', '1'), color = 'k', size = 20)

## Right Y-axis labels
pylab.text(1.05, 0.5, r'\bf{level set} $\phi$', {color: 'g', 'fontsize': 20},
    horizontalalignment = 'left',
    verticalalignment = 'center',
    rotation = 90,
    clip_on = False)
pylab.text(1.01, -0.02, "-1", {color: 'k', 'fontsize': 20})
pylab.text(1.01, 0.98, "1", {color: 'k', 'fontsize': 20})
pylab.text(1.01, 0.48, "0", {color: 'k', 'fontsize': 20})

## level set equations
pylab.text(0.1, 0.85, r'$|\nabla\phi| = 1,$ newline $\frac{\partial \phi}{\partial t} + U|\nabla \phi| = 0$

## phase field equations
```
pylab.text(0.2, 0.15, r'$\mathcal{F} = \int f\left( \phi, c \right) dV, \newline \frac{ \partial \phi } { \partial t } = -M_{ \phi } \frac{ \delta \mathcal{F} } { \delta \phi }$)

```python
## these went wrong in pdf in a previous version
pylab.text(-.9,.42,r'gamma: $\gamma$', {'color': 'r', 'fontsize': 20})
pylab.text(-.9,.36,r'Omega: $\Omega$', {'color': 'b', 'fontsize': 20})
pylab.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 84.213 pylab_examples example code: usetex_fonteffects.py

Usetex font effects

**Nimbus Roman No9 L (extended)**

**Nimbus Roman No9 L (condensed)**

**Nimbus Roman No9 L (slanted)**

**Nimbus Roman No9 L Italics (real italics for con**

**Nimbus Roman No9 L**

# This script demonstrates that font effects specified in your pdftex.map
# are now supported in pdf usetex.

```python
import matplotlib
matplotlib.rc('text', usetex=True)
import pylab

def setfont(font):
    return r'\font\a%s at 14pt\a % font'
```

84.213.  pylab_examples example code: usetex_fonteffects.py  2351
Matplotlib, Release 1.4.3

for y, font, text in zip(range(5),
    ['ptmr8r', 'ptmri8r', 'ptmro8r', 'ptmr8rn', 'ptmrr8re'],
    ['Nimbus Roman No9 L ' + x for x in
    ['', 'Italics (real italics for comparison)',
    '(slanted)', '(condensed)', '(extended)']]):
    pylab.text(0, y, setfont(font) + text)

pylab.ylim(-1, 5)
pylab.xlim(-0.2, 0.6)
pylab.setp(pylab.gca(), frame_on=False, xticks=(), yticks=())
pylab.title('Usetex font effects')
pylab.savefig('usetex_fonteffects.pdf')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

84.214 pylab_examples example code: vline_hline_demo.py

#!/usr/bin/env python

""
Small demonstration of the hlines and vlines plots.
""

from matplotlib import pyplot as plt
from numpy import sin, exp, absolute, pi, arange
from numpy.random import normal

def f(t):
    s1 = sin(2 * pi * t)
```python
e1 = exp(-t)
    return absolute((s1 * e1)) + .05

t = arange(0.0, 5.0, 0.1)
s = f(t)
nse = normal(0.0, 0.3, t.shape) * s

fig = plt.figure(figsize=(12, 6))
vax = fig.add_subplot(121)
hax = fig.add_subplot(122)

vax.plot(t, s + nse, 'b^')
vax.vlines(t, [0], s)
vax.set_xlabel('time (s)')
vax.set_title('Vertical lines demo')

hax.plot(s + nse, t, 'b^')
hax.hlines(t, [0], s, lw=2)
hax.set_xlabel('time (s)')
hax.set_title('Horizontal lines demo')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**84.215 pylab_examples example code: webapp_demo.py**

[source code]

```
#!/usr/bin/env python
# -*- noplot -*-
""
This example shows how to use the agg backend directly to create images, which may be of use to web application developers who want full control over their code without using the pyplot interface to manage figures, figure closing etc.

.. note::

    It is not necessary to avoid using the pyplot interface in order to create figures without a graphical front-end - simply setting the backend to "Agg" would be sufficient.

It is also worth noting that, because matplotlib can save figures to file-like object, matplotlib can also be used inside a cgi-script *without* needing to write a figure to disk.

""
```
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
import numpy as np

def make_fig():
    ""
    Make a figure and save it to "webagg.png".
    ""
    fig = Figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.plot([1, 2, 3], 'ro--', markersize=12, markerfacecolor='g')
    
    # make a translucent scatter collection
    x = np.random.rand(100)
    y = np.random.rand(100)
    area = np.pi * (10 * np.random.rand(100)) ** 2  # 0 to 10 point radiuses
    c = ax.scatter(x, y, area)
    c.set_alpha(0.5)
    
    # add some text decoration
    ax.set_title('My first image')
    ax.set_ylabel('Some numbers')
    ax.set_xticks((.2, .4, .6, .8))
    labels = ax.set_xticklabels(('Bill', 'Fred', 'Ted', 'Ed'))
    
    # To set object properties, you can either iterate over the
    # objects manually, or define you own set command, as in setapi
    # above.
    for label in labels:
        label.set_rotation(45)
        label.set_fontsize(12)

    FigureCanvasAgg(fig).print_png('webapp.png', dpi=150)

make_fig()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np

x, y = np.random.randn(2, 100)
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.xcorr(x, y, usevlines=True, maxlags=50, normed=True, lw=2)
ax1.grid(True)
ax1.axhline(0, color='black', lw=2)

ax2 = fig.add_subplot(212, sharex=ax1)
ax2.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)
ax2.grid(True)
ax2.axhline(0, color='black', lw=2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
84.217  pylab_examples example code: zorder_demo.py
The default drawing order for axes is patches, lines, text. This order is determined by the zorder attribute. The following defaults are set:

<table>
<thead>
<tr>
<th>Artist</th>
<th>Z-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch / PatchCollection</td>
<td>1</td>
</tr>
<tr>
<td>Line2D / LineCollection</td>
<td>2</td>
</tr>
<tr>
<td>Text</td>
<td>3</td>
</tr>
</tbody>
</table>

You can change the order for individual artists by setting the zorder. Any individual plot() call can set a value for the zorder of that particular item.

In the first subplot below, the lines are drawn above the patch collection from the scatter, which is the default.

In the subplot below, the order is reversed.

The second figure shows how to control the zorder of individual lines.

```python
from pylab import *
x = rand(20); y = rand(20)
```
```
subplot(211)
plot(x, y, 'r', lw=3)
scatter(x, y, s=120)

subplot(212)
plot(x, y, 'r', zorder=1, lw=3)
scatter(x, y, s=120, zorder=2)

# A new figure, with individually ordered items
x=frange(0,2*pi,npts=100)
figure()
plot(x, sin(x), linewidth=10, color='black', label='zorder=10', zorder = 10)  # on top
plot(x, cos(1.3*x), linewidth=10, color='red', label='zorder=1', zorder = 1)  # bottom
plot(x, sin(2.1*x), linewidth=10, color='green', label='zorder=3', zorder = 3)
axhline(0, linewidth=10, color='blue', label='zorder=2', zorder = 2)
1 = legend()
l.set_zorder(20)  # put the legend on top

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
85.1 shapes_and_collections example code: artist_reference.py

"""
Reference for matplotlib artists

This example displays several of matplotlib's graphics primitives (artists) drawn using matplotlib API. A full list of artists and the documentation is available at http://matplotlib.org/api/artist_api.html.
"""
```python
def label(xy, text):
    y = xy[1] - 0.15  # shift y-value for label so that it's below the artist
    plt.text(xy[0], y, text, ha="center", family='sans-serif', size=14)

fig, ax = plt.subplots()
# create 3x3 grid to plot the artists
grid = np.mgrid[0.2:0.8:3j, 0.2:0.8:3j].reshape(2, -1).T

patches = []

# add a circle
circle = mpatches.Circle(grid[0], 0.1, ec="none")
patches.append(circle)
label(grid[0], "Circle")

# add a rectangle
rect = mpatches.Rectangle(grid[1] - [0.025, 0.05], 0.05, 0.1, ec="none")
patches.append(rect)
label(grid[1], "Rectangle")

# add a wedge
wedge = mpatches.Wedge(grid[2], 0.1, 30, 270, ec="none")
patches.append(wedge)
label(grid[2], "Wedge")

# add a Polygon
polygon = mpatches.RegularPolygon(grid[3], 5, 0.1)
patches.append(polygon)
label(grid[3], "Polygon")

# add an ellipse
ellipse = mpatches.Ellipse(grid[4], 0.2, 0.1)
patches.append(ellipse)
label(grid[4], "Ellipse")

# add an arrow
arrow = mpatches.Arrow(grid[5, 0]-0.05, grid[5, 1]-0.05, 0.1, 0.1, width=0.1)
patches.append(arrow)
label(grid[5], "Arrow")
```
# add a path patch
Path = mpath.Path
path_data = [
    (Path.MOVETO, [0.018, -0.11]),
    (Path.CURVE4, [-0.031, -0.051]),
    (Path.CURVE4, [-0.115, 0.073]),
    (Path.CURVE4, [-0.03, 0.073]),
    (Path.LINETO, [-0.011, 0.039]),
    (Path.CURVE4, [0.043, 0.121]),
    (Path.CURVE4, [0.075, -0.005]),
    (Path.CURVE4, [0.035, -0.027]),
    (Path.CLOSEPOLY, [0.018, -0.11]),
]
codes, verts = zip("path_data")
path = mpath.Path(verts + grid[6], codes)
patch = mpatches.PathPatch(path)
patches.append(patch)
label(grid[6], "PathPatch")

# add a fancy box
fancybox = mpatches.FancyBboxPatch(
    grid[7] - [0.025, 0.05], 0.05, 0.1,
    boxstyle=mpatches.BoxStyle("Round", pad=0.02))
patches.append(fancybox)
label(grid[7], "FancyBoxPatch")

# add a line
x, y = np.array([[-0.06, 0.0, 0.1], [0.05, -0.05, 0.05]])
line = mlines.Line2D(x + grid[8, 0], y + grid[8, 1], lw=5., alpha=0.3)
label(grid[8], "Line2D")

colors = np.linspace(0, 1, len(patches))
collection = PatchCollection(patches, cmap=plt.cm.hsv, alpha=0.3)
collection.set_array(np.array(colors))
ax.add_collection(collection)
ax.add_line(line)

plt.subplots_adjust(left=0, right=1, bottom=0, top=1)
plt.axis('equal')
plt.axis('off')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demonstration of a PathPatch object.

```python
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
Path = mpath.Path
path_data = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
]
```
codes, verts = zip(*path_data)
path = mpath.Path(verts, codes)
patch = mpatches.PathPatch(path, facecolor='r', alpha=0.5)
ax.add_patch(patch)

# plot control points and connecting lines
x, y = zip(*path.vertices)
line, = ax.plot(x, y, 'go-')

ax.grid()
ax.axis('equal')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

85.3 shapes_and_collections example code: scatter_demo.py

""
Simple demo of a scatter plot.
""
import numpy as np
import matplotlib.pyplot as plt
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radiuses

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Plot demonstrating the integral as the area under a curve.

Although this is a simple example, it demonstrates some important tweaks:

* A simple line plot with custom color and line width.
* A shaded region created using a Polygon patch.
* A text label with mathtext rendering.
def func(x):
    return (x - 3) * (x - 5) * (x - 7) + 85

a, b = 2, 9  # integral limits
x = np.linspace(0, 10)
y = func(x)

fig, ax = plt.subplots()
plt.plot(x, y, '
', linewidth=2)
plt.ylim(ymin=0)
# Make the shaded region
ix = np.linspace(a, b)
iy = func(ix)
verts = [(a, 0)] + list(zip(ix, iy)) + [(b, 0)]
poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
ax.add_patch(poly)
plt.text(0.5 * (a + b), 30, r'$\int_a^b f(x)\mathrm{d}x$', horizontalalignment='center', fontsize=20)
plt.figtext(0.9, 0.05, '$x$')
plt.figtext(0.1, 0.9, '$y$')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.xaxis.set_ticks_position('bottom')

ax.set_xticks((a, b))
ax.set_xticklabels(('a', 'b'))
ax.set_yticks([])

plt.show()
86.2 showcase example code: xkcd.py

"Stove Ownership" from xkcd by Randall Monroe

THE DAY I REALIZED I COULD COOK BACON WHENEVER I WANTED
import matplotlib.pyplot as plt
import numpy as np

with plt.xkcd():
    # Based on "Stove Ownership" from XKCD by Randall Monroe
    # http://xkcd.com/418/

    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.2, 0.8, 0.7])
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    plt.xticks([])
    plt.yticks([])
    ax.set_ylim([-30, 10])

    data = np.ones(100)
    data[70:] -= np.arange(30)

    plt.annotate('THE DAY I REALIZED
I COULD COOK BACON
WHENEVER I WANTED',
                xy=(70, 1),
                arrowprops=dict(arrowstyle='->'),
                xytext=(15, -10))

    plt.plot(data)
    plt.xlabel('time')
plt.ylabel('my overall health')
fig.text(0.5, 0.05, 
    '"Stove Ownership" from xkcd by Randall Monroe', 
    ha='center')

# Based on "The Data So Far" from XKCD by Randall Monroe
# http://xkcd.com/373/

fig = plt.figure()
ax = fig.add_axes([0.1, 0.2, 0.8, 0.7])
ax.bar([-0.125, 1.0-0.125], [0, 100], 0.25)
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.set_xticks([0, 1])
ax.set_xlim([-0.5, 1.5])
ax.set_yticks([])
ax.set_xlabel('CONFIRMED BY\nEXPERIMENT', 'REFUTED BY\nEXPERIMENT')
plt.title("CLAIMS OF SUPERNATURAL POWERS")

fig.text(0.5, 0.05, 
    '"The Data So Far" from xkcd by Randall Monroe', 
    ha='center')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
87.1 specialty_plots example code: hinton_demo.py

""
Demo of a function to create Hinton diagrams.

Hinton diagrams are useful for visualizing the values of a 2D array (e.g. a weight matrix): Positive and negative values are represented by white and black squares, respectively, and the size of each square represents the magnitude of each value.
"""
import numpy as np
import matplotlib.pyplot as plt

def hinton(matrix, max_weight=None, ax=None):
    """Draw Hinton diagram for visualizing a weight matrix."""
    ax = ax if ax is not None else plt.gca()

    if not max_weight:
        max_weight = 2**np.ceil(np.log(np.abs(matrix).max()) / np.log(2))

    ax.patch.set_facecolor('gray')
    ax.set_aspect('equal', 'box')
    ax.xaxis.set_major_locator(plt.NullLocator())
    ax.yaxis.set_major_locator(plt.NullLocator())

    for (x, y), w in np.ndenumerate(matrix):
        color = 'white' if w > 0 else 'black'
        size = np.sqrt(np.abs(w))
        rect = plt.Rectangle([x - size / 2, y - size / 2], size, size,
                              facecolor=color, edgecolor=color)
        ax.add_patch(rect)

    ax.autoscale_view()
    ax.invert_yaxis()

if __name__ == '__main__':
    hinton(np.random.rand(20, 20) - 0.5)
    plt.show()
88.1 statistics example code: boxplot_demo.py
I never said they’d be pretty

Custom boxprops

Custom medianprops

and flierprops

whis="range"

Custom mean

as point

Custom mean

as line

whis=[15, 85]

# percentiles

Demo of the new boxplot functionality

```
import numpy as np
import matplotlib.pyplot as plt

# fake data
np.random.seed(937)
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)
labels = list('ABCD')
fs = 10  # fontsize

# demonstrate how to toggle the display of different elements:
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6,6))
axes[0, 0].boxplot(data, labels=labels)
```
axes[0, 0].set_title('Default', fontsize=fs)

axes[0, 1].boxplot(data, labels=labels, showmeans=True)
axes[0, 1].set_title('showmeans=True', fontsize=fs)

axes[0, 2].boxplot(data, labels=labels, showmeans=True, meanline=True)
axes[0, 2].set_title('showmeans=True, meanline=True', fontsize=fs)

axes[1, 0].boxplot(data, labels=labels, showbox=False, showcaps=False)
axes[1, 0].set_title('Tufte Style (showbox=False, showcaps=False)', fontsize=fs)

axes[1, 1].boxplot(data, labels=labels, notch=True, bootstrap=10000)
axes[1, 1].set_title('notch=True, bootstrap=10000', fontsize=fs)

axes[1, 2].boxplot(data, labels=labels, showfliers=False)
axes[1, 2].set_title('showfliers=False', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.subplots_adjust(hspace=0.4)
plt.show()

# demonstrate how to customize the display different elements:
boxprops = dict(linestyle='--', linewidth=3, color='darkgoldenrod')
flierprops = dict(marker='o', markerfacecolor='green', markersize=12,
                   linestyle='none')
medianprops = dict(linestyle='-.', linewidth=2.5, color='firebrick')
meanpointprops = dict(marker='D', markeredgecolor='black',
                       markerfacecolor='firebrick')
meanlineprops = dict(linestyle='--', linewidth=2.5, color='purple')

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6,6))
axes[0, 0].boxplot(data, boxprops=boxprops)
axes[0, 0].set_title('Custom boxprops', fontsize=fs)

axes[0, 1].boxplot(data, flierprops=flierprops, medianprops=medianprops)
axes[0, 1].set_title('Custom medianprops and flierprops', fontsize=fs)

axes[0, 2].boxplot(data, whis='range')
axes[0, 2].set_title('whis='range', fontsize=fs)

axes[1, 0].boxplot(data, meanprops=meanpointprops, meanline=False, showmeans=True)
axes[1, 0].set_title('Custom mean\ nas point', fontsize=fs)

axes[1, 1].boxplot(data, meanprops=meanlineprops, meanline=True, showmeans=True)
axes[1, 1].set_title('Custom mean\ nas line', fontsize=fs)

axes[1, 2].boxplot(data, whis=[15, 85])
axes[1, 2].set_title('whis=[15, 85]\#percentiles', fontsize=fs)
for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.suptitle("I never said they'd be pretty")
fig.subplots_adjust(hspace=0.4)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 88.2 statistics example code: bxp_demo.py

![Box plots with different options](image-url)
I never said they'd be pretty

Custom boxprops

Custom medianprops and flierprops

Custom mean as point

Custom mean as line

Demo of the new boxplot drawer function

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

# fake data
np.random.seed(937)
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)
labels = list('ABCD')

# compute the boxplot stats
stats = cbook.boxplot_stats(data, labels=labels, bootstrap=10000)
# After we've computed the stats, we can go through and change anything.
# Just to prove it, I'll set the median of each set to the median of all
# the data, and double the means
for n in range(len(stats)):
    stats[n]["med"] = np.median(data)
    stats[n]["mean"] *= 2

print(stats[0].keys())

fs = 10  # fontsize

# demonstrate how to toggle the display of different elements:
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6,6))
axes[0, 0].bxp(stats)
axes[0, 0].set_title('Default', fontsize=fs)
axes[0, 1].bxp(stats, showmeans=True)
axes[0, 1].set_title('showmeans=True', fontsize=fs)
axes[0, 2].bxp(stats, showmeans=True, meanline=True)
axes[0, 2].set_title('showmeans=True, meanline=True', fontsize=fs)
axes[1, 0].bxp(stats, showbox=False, showcaps=False)
axes[1, 0].set_title('Tufte Style
(showbox=False, nshowcaps=False)', fontsize=fs)
axes[1, 1].bxp(stats, showfliers=False)
axes[1, 1].set_title('showfliers=False', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])
fig.subplots_adjust(hspace=0.4)
plt.show()

# demonstrate how to customize the display different elements:
boxprops = dict(linestyle='--', linewidth=3, color='darkgoldenrod')
flierprops = dict(marker='o', markerfacecolor='green', markersize=12,
                 linestyle='none')
medianprops = dict(linestyle='-.', linewidth=2.5, color='firebrick')
meanpointprops = dict(marker='D', markeredgecolor='black',
                      markerfacecolor='firebrick')
meanlineprops = dict(linestyle='--', linewidth=2.5, color='purple')

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(6,6))
axes[0, 0].bxp(stats, boxprops=boxprops)
axes[0, 0].set_title('Custom boxprops', fontsize=fs)
axes[0, 1].bxp(stats, flierprops=flierprops, medianprops=medianprops)
axes[0, 1].set_title('Custom medianprops\nand flierprops', fontsize=fs)
axes[1, 0].bxp(stats, meanprops=meanpointprops, meanline=False, showmeans=True)
axes[1, 0].set_title('Custom mean as point', fontsize=fs)

axes[1, 1].bxp(stats, meanprops=meanlineprops, meanline=True, showmeans=True)
axes[1, 1].set_title('Custom mean as line', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.suptitle("I never said they'd be pretty")
fig.subplots_adjust(hspace=0.4)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

88.3 statistics example code: errorbar_demo.py

""
Demo of the errorbar function.
"""
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

plt.errorbar(x, y, xerr=0.2, yerr=0.4)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

88.4 statistics example code: errorbar_demo_features.py

```
Demo of errorbar function with different ways of specifying error bars.

Errors can be specified as a constant value (as shown in `errorbar_demo.py`),
or as demonstrated in this example, they can be specified by an N x 1 or 2 x N,
where N is the number of data points.

N x 1:
```
Error varies for each point, but the error values are symmetric (i.e. the lower and upper values are equal).

2 x N:
Error varies for each point, and the lower and upper limits (in that order) are different (asymmetric case)

In addition, this example demonstrates how to use log scale with errorbar.

```python
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)
# example error bar values that vary with x-position
error = 0.1 + 0.2 * x
# error bar values w/ different -/+ errors
lower_error = 0.4 * error
upper_error = error
asymmetric_error = [lower_error, upper_error]

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True)
ax0.errorbar(x, y, yerr=error, fmt='-o')
ax0.set_title('variable, symmetric error')

ax1.errorbar(x, y, xerr=asymmetric_error, fmt='o')
ax1.set_title('variable, asymmetric error')
ax1.set_yscale('log')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of the errorbar function, including upper and lower limits

```python
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.5, 5.5, 0.5)
y = np.exp(-x)
xerr = 0.1
yerr = 0.2
ls = 'dotted'

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

# standard error bars
plt.errorbar(x, y, xerr=xerr, yerr=yerr, ls=ls, color='blue')

# including upper limits
uplims = np.zeros(x.shape)
uplims[[1, 5, 9]] = True
```
```python
plt.errorbar(x, y+.5, xerr=xerr, yerr=yerr, uplims=uplims, ls=ls,
    color='green')

# including lower limits
lolims = np.zeros(x.shape)
lolims[2, 4, 8] = True
plt.errorbar(x, y+1.0, xerr=xerr, yerr=yerr, lolims=lolims, ls=ls,
    color='red')

# including upper and lower limits
plt.errorbar(x, y+1.5, marker='o', ms=8, xerr=xerr, yerr=yerr,
    lolims=lolims, uplims=uplims, ls=ls, color='magenta')

# including xlower and xupper limits
xerr = 0.2
yerr = np.zeros(x.shape) + 0.2
yerr[3, 6] = 0.3
xlolims = lolims
xuplims = uplims
lolims = np.zeros(x.shape)
uplims = np.zeros(x.shape)
lolims[6] = True
uplims[3] = True
plt.errorbar(x, y+2.1, marker='o', ms=8, xerr=xerr, yerr=yerr,
    xlolims=xlolims, xuplims=xuplims, uplims=uplims, lolims=lolims,
    ls='none', mec='blue', capsize=0, color='cyan')

ax.set_xlim((0, 5.5))
ax.set_title('Errorbar upper and lower limits')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of the histogram (hist) function used to plot a cumulative distribution.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import mlab

mu = 200
sigma = 25
n_bins = 50
x = mu + sigma * np.random.randn(10000)

n, bins, patches = plt.hist(x, n_bins, normed=1,
                            histtype='step', cumulative=True)

y = mlab.normpdf(bins, mu, sigma).cumsum()
y /= y[-1]
plt.plot(bins, y, 'k--', linewidth=1.5)
```
# Overlay a reversed cumulative histogram.
plt.hist(x, bins=bins, normed=1, histtype='step', cumulative=-1)

plt.grid(True)
plt.ylim(0, 1.05)
plt.title('cumulative step')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

88.7 statistics example code: histogram_demo_features.py

Demo of the histogram (hist) function with a few features.

In addition to the basic histogram, this demo shows a few optional features:

* Setting the number of data bins
* The `normed` flag, which normalizes bin heights so that the integral of the histogram is 1. The resulting histogram is a probability density.
* Setting the face color of the bars
* Setting the opacity (alpha value).

```
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

# example data
mu = 100  # mean of distribution
sigma = 15  # standard deviation of distribution
x = mu + sigma * np.random.randn(10000)

num_bins = 50
# the histogram of the data
n, bins, patches = plt.hist(x, num_bins, normed=1, facecolor='green', alpha=0.5)
# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
plt.plot(bins, y, 'r--')
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'_histogram of IQ: $\mu=100$, $\sigma=15$')

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

88.8 statistics example code: histogram_demo_histtypes.py
Demo of the histogram (hist) function with different `histtype` settings.

- Histogram with step curve that has a color fill.
- Histogram with with unequal bin widths.

```python
import numpy as np
import matplotlib.pyplot as plt

mu = 200
sigma = 25
x = mu + sigma*np.random.randn(10000)

fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(8, 4))

ax0.hist(x, 20, normed=1, histtype='stepfilled', facecolor='g', alpha=0.75)
ax0.set_title('stepfilled')

# Create a histogram by providing the bin edges (unequally spaced).
bins = [100, 150, 180, 195, 205, 220, 250, 300]
ax1.hist(x, bins, normed=1, histtype='bar', rwidth=0.8)
ax1.set_title('unequal bins')

plt.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of the histogram (hist) function with multiple data sets.

Plot histogram with multiple sample sets and demonstrate:

* Use of legend with multiple sample sets
* Stacked bars
* Step curve with a color fill
* Data sets of different sample sizes

```
import numpy as np
import matplotlib.pyplot as plt

n_bins = 10
x = np.random.randn(1000, 3)

fig, axes = plt.subplots(nrows=2, ncols=2)
ax0, ax1, ax2, ax3 = axes.flat

colors = ['red', 'tan', 'lime']
ax0.hist(x, n_bins, normed=1, histtype='bar', color=colors, label=colors)
```
ax0.legend(prop={'size': 10})
ax0.set_title('bars with legend')

ax1.hist(x, n_bins, normed=1, histtype='bar', stacked=True)
ax1.set_title('stacked bar')

ax2.hist(x, n_bins, histtype='step', stacked=True, fill=True)
ax2.set_title('stepfilled')

# Make a multiple-histogram of data-sets with different length.
x_multi = [np.random.randn(n) for n in [10000, 5000, 2000]]
ax3.hist(x_multi, n_bins, histtype='bar')
ax3.set_title('different sample sizes')

plt.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of the new violinplot functionality

```python
import random
import numpy as np
import matplotlib.pyplot as plt

# fake data
fs = 10  # fontsize
pos = [1, 2, 4, 5, 7, 8]
data = [np.random.normal(size=100) for i in pos]
```
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6,6))

axes[0, 0].violinplot(data, pos, points=20, widths=0.1,
                      showmeans=True, showextrema=True, showmedians=True)
axes[0, 0].set_title('Custom violinplot 1', fontsize=fs)

axes[0, 1].violinplot(data, pos, points=40, widths=0.3,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method='silverman')
axes[0, 1].set_title('Custom violinplot 2', fontsize=fs)

axes[0, 2].violinplot(data, pos, points=60, widths=0.5, showmeans=True,
                      showextrema=True, showmedians=True,
                      bw_method=0.5)
axes[0, 2].set_title('Custom violinplot 3', fontsize=fs)

axes[1, 0].violinplot(data, pos, points=80, vert=False, widths=0.7,
                      showmeans=True, showextrema=True, showmedians=True)
axes[1, 0].set_title('Custom violinplot 4', fontsize=fs)

axes[1, 1].violinplot(data, pos, points=100, vert=False, widths=0.9,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method='silverman')
axes[1, 1].set_title('Custom violinplot 5', fontsize=fs)

axes[1, 2].violinplot(data, pos, points=200, vert=False, widths=1.1,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method=0.5)
axes[1, 2].set_title('Custom violinplot 6', fontsize=fs)

for ax in axes.flatten():
    ax.set_yticklabels([])

fig.suptitle("Violin Plotting Examples")
fig.subplots_adjust(hspace=0.4)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
89.1 style_sheets example code: plot_bmh.py

```python
from numpy.random import beta
import matplotlib.pyplot as plt
plt.style.use('bmh')
plt.style.use('bmh')
```

This example demonstrates the "bmh" style, which is the design used in the Bayesian Methods for Hackers online book.
def plot_beta_hist(a, b):
    plt.hist(beta(a, b, size=10000), histtype="stepfilled",
             bins=25, alpha=0.8, normed=True)
    return

plot_beta_hist(10, 10)
plot_beta_hist(4, 12)
plot_beta_hist(50, 12)
plot_beta_hist(6, 55)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.2 style_sheets example code: plot_dark_background.py

"""
This example demonstrates the "dark_background" style, which uses white for
elements that are typically black (text, borders, etc). Note, however, that not
all plot elements default to colors defined by an rc parameter.
"""
```python
import numpy as np
import matplotlib.pyplot as plt

plt.style.use('dark_background')

L = 6
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.color_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    plt.plot(x, np.sin(x + s), 'o-')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('title')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**89.3 style_sheets example code: plot_fivethirtyeight.py**
This shows an example of the "fivethirtyeight" styling, which tries to replicate the styles from FiveThirtyEight.com.

```
from matplotlib import pyplot as plt
import numpy as np

x = np.linspace(0, 10)

with plt.style.context('fivethirtyeight'):
    plt.plot(x, np.sin(x) + x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 2 * x + np.random.randn(50))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.4 style_sheets example code: plot_ggplot.py
This example demonstrates the "ggplot" style, which adjusts the style to emulate ggplot_ (a popular plotting package for R_).

These settings were shamelessly stolen from [1]_ (with permission).


.. _ggplot: http://had.co.nz/ggplot/
.. _R: http://www.r-project.org/

import numpy as np
import matplotlib.pyplot as plt

plt.style.use('ggplot')

fig, axes = plt.subplots(nrows=2, ncols=2)
ax1, ax2, ax3, ax4 = axes.ravel()

# scatter plot (Note: `plt.scatter` doesn't use default colors)
x, y = np.random.normal(size=(2, 200))
ax1.plot(x, y, 'o')

# sinusoidal lines with colors from default color cycle
L = 2*np.pi
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.color_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    ax2.plot(x, np.sin(x + s), '-')
ax2.margins(0)

# bar graphs
x = np.arange(5)
y1, y2 = np.random.randint(25, size=(2, 5))
width = 0.25
ax3.bar(x, y1, width)
ax3.bar(x+width, y2, width, color=plt.rcParams['axes.color_cycle'][2])
ax3.set_xticks(x+width)
ax3.set_xticklabels(['a', 'b', 'c', 'd', 'e'])

# circles with colors from default color cycle
for i, color in enumerate(plt.rcParams['axes.color_cycle']):
    xy = np.random.normal(size=2)
    ax4.add_patch(plt.Circle(xy, radius=0.3, color=color))
ax4.axis('equal')
ax4.margins(0)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This example demonstrates the "grayscale" style sheet, which changes all colors that are defined as rc parameters to grayscale. Note, however, that not all plot elements default to colors defined by an rc parameter.

```
import numpy as np
import matplotlib.pyplot as plt

def color_cycle_example(ax):
    L = 6
    x = np.linspace(0, L)
    ncolors = len(plt.rcParams['axes.color_cycle'])
    shift = np.linspace(0, L, ncolors, endpoint=False)
    for s in shift:
        ax.plot(x, np.sin(x + s), 'o-')

def image_and_patch_example(ax):
    ax.imshow(np.random.random(size=(20, 20)), interpolation='none')
    c = plt.Circle((5, 5), radius=5, label='patch')
    ax.add_patch(c)
```
plt.style.use('grayscale')

fig, (ax1, ax2) = plt.subplots(ncols=2)

color_cycle_example(ax1)
image_and_patch_example(ax2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of how to display two scales on the left and right y axis.

This example uses the Fahrenheit and Celsius scales.

```python
import matplotlib.pyplot as plt
import numpy as np
```
def fahrenheit2celsius(temp):
    """
    Returns temperature in Celsius.
    """
    return (5. / 9.) * (temp - 32)

def convert_ax_c_to_celsius(ax_f):
    """
    Update second axis according with first axis.
    """
    y1, y2 = ax_f.get_ylim()
    ax_c.set_ylim(fahrenheit2celsius(y1), fahrenheit2celsius(y2))
    ax_c.figure.canvas.draw()

fig, ax_f = plt.subplots()
ax_c = ax_f.twinx()

# automatically update ylim of ax2 when ylim of ax1 changes.
ax_f.callbacks.connect("ylim_changed", convert_ax_c_to_celsius)
ax_f.plot(np.linspace(-40, 120, 100))
ax_f.set_xlim(0, 100)
ax_f.set_title('Two scales: Fahrenheit and Celsius')
ax_f.set_ylabel('Fahrenheit')
ax_c.set_ylabel('Celsius')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Simple demo with multiple subplots.

```python
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'yo-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')
```

---

90.2 subplots_axes_and_figures example code: subplot_demo.py

---

A tale of 2 subplots

---

```
Simple demo with multiple subplots.
```

---

```python
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'yo-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')
```
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
91.1 tests example code: backend_driver.py

[source code]

#!/usr/bin/env python

from __future__ import print_function, division

"""
This is used to drive many of the examples across the backends, for
regression testing, and comparing backend efficiency.

You can specify the backends to be tested either via the --backends
switch, which takes a comma-separated list, or as separate arguments,
e.g.

    python backend_driver.py agg ps

would test the agg and ps backends. If no arguments are given, a
default list of backends will be tested.

Interspersed with the backend arguments can be switches for the Python
interpreter executing the tests. If entering such arguments causes an
option parsing error with the driver script, separate them from driver
switches with a --.
""

import os
import time
import sys
import glob
from optparse import OptionParser

import matplotlib.rcsetup as rcsetup
from matplotlib.cbook import Bunch, dedent

all_backends = list(rcsetup.all_backends)  # to leave the original list alone
# actual physical directory for each dir
dirs = dict(files=os.path.join('..', 'lines_bars_and_markers'),
            shapes=os.path.join('..', 'shapes_and_collections'),
            images=os.path.join('..', 'images_contours_and_fields'),
            pie=os.path.join('..', 'pie_and_polar_charts'),
            text=os.path.join('..', 'text_labels_and_annotations'),
            ticks=os.path.join('..', 'ticks_and_spines'),
            subplots=os.path.join('..', 'subplots_axes_and_figures'),
            specialty=os.path.join('..', 'specialty_plots'),
            showcase=os.path.join('..', 'showcase'),
            pylab = os.path.join('..', 'pylab_examples'),
            api = os.path.join('..', 'api'),
            units = os.path.join('..', 'units'),
            mplot3d = os.path.join('..', 'mplot3d'))

# files in each dir
files = dict()

files['lines'] = [
    'barh_demo.py',
    'fill_demo.py',
    'fill_demo_features.py',
    'line_demo_dash_control.py',
    'line_styles_reference.py',
    'scatter_with_legend.py'
]

files['shapes'] = [
    'path_patch_demo.py',
    'scatter_demo.py',
]

files['colors'] = [
    'color_cycle_demo.py',
]

files['images'] = [
    'imshow_demo.py',
]

files['statistics'] = [
    'errorbar_demo.py',
    'errorbar_demo_features.py',
    'histogram_demo_cumulative.py',
    'histogram_demo_features.py',
    'histogram_demo_histtypes.py',
    'histogram_demo_multihist.py',
]

files['pie'] = [
    'pie_demo.py',
    'polar_bar_demo.py',
    'polar_scatter_demo.py',
]
files['text_labels_and_annotations'] = [
    'text_demo_fontdict.py',
    'unicode_demo.py',
]

files['ticks_and_spines'] = [
    'spines_demo_bounds.py',
    'ticklabels_demo_rotation.py',
]

files['subplots_axes_and_figures'] = [
    'subplot_demo.py',
]

files['showcase'] = [
    'integral_demo.py',
]

files['pylab'] = [
    'accented_text.py',
    'alignment_test.py',
    'annotation_demo.py',
    'annotation_demo.py',
    'annotation_demo2.py',
    'annotation_demo2.py',
    'anscombe.py',
    'arctest.py',
    'arrow_demo.py',
    'axes_demo.py',
    'axes_props.py',
    'axhspan_demo.py',
    'axis_equal_demo.py',
    'bar_stacked.py',
    'barb_demo.py',
    'barchart_demo.py',
    'barcode_demo.py',
    'boxplot_demo.py',
    'broken_barh.py',
    'clippedline.py',
    'cohere_demo.py',
    'color_by_yvalue.py',
    'color_demo.py',
    'colorbar_tick_labelling_demo.py',
    'contour_demo.py',
    'contour_image.py',
    'contour_label_demo.py',
    'contourf_demo.py',
    'contourf_log.py',
    'coords_demo.py',
    'coords_report.py',
    'csd_demo.py',
]
'cursor_demo.py',
'custom_cmap.py',
'custom_figure_class.py',
'custom_ticker1.py',
'customize_rc.py',
'dashpointlabel.py',
'date_demo1.py',
'date_demo2.py',
'date_demo_convert.py',
'date_demo_rrule.py',
'date_index_formatter.py',
'dolphin.py',
'ellipse_collection.py',
'ellipse_demo.py',
'ellipse_rotated.py',
'equal_aspect_ratio.py',
'errorbar_limits.py',
'fancybox_demo.py',
'fancybox_demo2.py',
'fancytextbox_demo.py',
'figimage_demo.py',
'figlegend_demo.py',
'figure_title.py',
'fill_between_demo.py',
'fill_spiral.py',
'finance_demo.py',
'findobj_demo.py',
'fonts_demo.py',
'fonts_demo_kw.py',
'ganged_plots.py',
'geo_demo.py',
'gradient_bar.py',
'griddata_demo.py',
'hatch_demo.py',
'hexbin_demo.py',
'hexbin_demo2.py',
'hist_colormapped.py',
'vline_hline_demo.py',
'image_clip_path.py',
'image_demo.py',
'image_demo2.py',
'image_interp.py',
'image_masked.py',
'image_nonuniform.py',
'image_origin.py',
'image_slices_viewer.py',
'interp_demo.py',
'invert_axes.py',
'layer_images.py',
'legend_demo2.py',
'legend_demo3.py',

'line_collection.py',
'line_collection2.py',
'log_bar.py',
'log_demo.py',
'log_test.py',
'major_minor_demo1.py',
'major_minor_demo2.py',
'manual_axis.py',
'masked_demo.py',
'mathtext_demo.py',
'mathtext_examples.py',
'matplotlib_icon.py',
'matshow.py',
'mri_demo.py',
'mri_with_eeg.py',
'multi_image.py',
'multiline.py',
'multiple_figs_demo.py',
'nan_test.py',
'newscalarformatter_demo.py',
'pcolor_demo.py',
'pcolor_log.py',
'pcolor_small.py',
'pie_demo2.py',
'plotfile_demo.py',
'polar_demo.py',
'polar_legend.py',
'psd_demo.py',
'psd_demo2.py',
'psd_demo3.py',
'quadmesh_demo.py',
'quiver_demo.py',
'scatter_custom_symbol.py',
'scatter_demo2.py',
'scatter_masked.py',
'scatter_profile.py',
'scatter_star_poly.py',
'set_and_get.py',
'shared_axis_across_figures.py',
'shared_axis_demo.py',
'simple_plot.py',
'specgram_demo.py',
'spine_placement_demo.py',
'spy_demos.py',
'startup_plot.py',
'step_demo.py',
'stix_fonts_demo.py',
'stock_demo.py',
'subplots_adjust.py',
'symlog_demo.py',
'table_demo.py',
'text_handles.py',
'text_rotation.py',

91.1. tests example code: backend_driver.py
files['api'] = [
    'agg_oo.py',
    'barchart_demo.py',
    'bbox_intersect.py',
    'collections_demo.py',
    'colorbar_only.py',
    'custom_projection_example.py',
    'custom_scale_example.py',
    'date_demo.py',
    'date_index_formatter.py',
    'donut_demo.py',
    'font_family_rc.py',
    'image_zcoord.py',
    'joinstyle.py',
    'legend_demo.py',
    'line_with_text.py',
    'logo2.py',
    'mathtext_asarray.py',
    'patch_collection.py',
    'quad_bezier.py',
    'scatter_piecharts.py',
    'span_regions.py',
    'two_scales.py',
    'unicode_minus.py',
    'watermark_image.py',
    'watermark_text.py',
]

files['units'] = [
    'annotate_with_units.py',
    '#artist_tests.py', # broken, fixme
    'bar_demo2.py',
    '#bar_unit_demo.py', # broken, fixme
    '#ellipse_with_units.py', # broken, fixme
    'radian_demo.py',
    'units_sample.py',
    '#units_scatter.py', # broken, fixme
]

files['mplot3d'] = [
    '2dcollections3d_demo.py',
    'bars3d_demo.py',
    'contour3d_demo.py',
    'contour3d_demo2.py',
    'contourf3d_demo.py',
]
Matplotlib, Release 1.4.3

'ex3d_demo.py',
'polys3d_demo.py',
'scatter3d_demo.py',
'surface3d_demo.py',
'surface3d_demo2.py',
'text3d_demo.py',
'wire3d_demo.py',
]

# dict from dir to files we know we don't want to test (e.g., examples
# not using pyplot, examples requiring user input, animation examples,
# examples that may only work in certain environs (usetex examples?),
# examples that generate multiple figures

excluded = {
    'pylab': ['__init__.py', 'toggle_images.py'],
    'units': ['__init__.py', 'date_support.py'],
}

def report_missing(dir, flist):
    # report the py files in dir that are not in flist'
    globstr = os.path.join(dir, '*.py')
    fnames = glob.glob(globstr)

    pyfiles = set([os.path.split(fullpath)[-1] for fullpath in set(fnames)])

    exclude = set(excluded.get(dir, []))
    flist = set(flist)
    missing = list(pyfiles-flist-exclude)
    missing.sort()
    if missing:
        print('%s files not tested: %s' % (dir, ', '.join(missing)))

def report_all_missing(directories):
    for f in directories:
        report_missing(dirs[f], files[f])

# tests known to fail on a given backend

failbackend = dict(
    svg = ('tex_demo.py', ),
    agg = ('hyperlinks.py', ),
    pdf = ('hyperlinks.py', ),
    ps = ('hyperlinks.py', ),
)

try:
    import subprocess
    def run(arglist):
        try:
            ret = subprocess.call(arglist)
except KeyboardInterrupt:
    sys.exit()
else:
    return ret
except ImportError:

def run(arglist):
    os.system(' '.join(arglist))

def drive(backend, directories, python=['python'], switches=[]):
    exclude = failbackend.get(backend, [])

    # Clear the destination directory for the examples
    path = backend
    if os.path.exists(path):
        import glob
        for fname in os.listdir(path):
            os.unlink(os.path.join(path, fname))
    else:
        os.mkdir(backend)

    failures = []

    testcases = [os.path.join(dirs[d], fname)
                 for d in directories
                 for fname in files[d]]

    for fullpath in testcases:
        print ('%driving %-40s' % (fullpath)),
        sys.stdout.flush()
        fp, f = os.path.split(fullpath)

        if f in exclude:
            print ('%SSkipping %s, known to fail on backend: %s' % (backend))
            continue

        basename, ext = os.path.splitext(f)
        out = os.path.join(fp, basename)
        tmpfname = '_tmp_%s.py' % basename
        tmpfile = open(tmpfname, 'w')

        future_imports = 'from __future__ import division, print_function'

        for line in open(fullpath):
            line_lstrip = line.lstrip()
            if line_lstrip.startswith('#'):
                tmpfile.write(line)
            elif 'unicode_literals' in line:
                future_imports = future_imports + ', unicode_literals'

        tmpfile.writelines((
            future_imports + '
',
            'import sys\n',
            'sys.path.append("%s")\n' % fp.replace('\\', '\\\\'),
            'import matplotlib\n',
            'matplotlib.use("%s")\n' % backend,

    print('%SPassed %s examples out of %s.' % (backend, len(testcases), failures))

    return True

    except: pass

'from pylab import savefig
',
'import numpy
',
'numpy.seterr(invalid="ignore")
',
)

for line in open(fullpath):
    line_lstrip = line.lstrip()
    if (line_lstrip.startswith('from __future__ import') or
        line_lstrip.startswith('matplotlib.use') or
        line_lstrip.startswith('savefig') or
        line_lstrip.startswith('show')):
        continue
    tmpfile.write(line)
    if backend in rcsetup.interactive_bk:
        tmpfile.write('show()')
    else:
        tmpfile.write('%savefig(r"%s", dpi=150)', % outfile)

    tmpfile.close()
start_time = time.time()
program = [x % {'name': basename} for x in python]
ret = run(program + [tmpfile_name] + switches)
end_time = time.time()
print('%s %s ((end_time - start_time), ret))
#os.system('%s %s %s' % (python, tmpfile_name, ' '.join(switches)))
    os.remove(tmpfile_name)
if ret:
    failures.append(fullpath)
return failures
def parse_options():
    doc = (__doc__ and __doc__.split('

')) or " "
    op = OptionParser(description=doc[0].strip(),
                       usage='%prog [options] [--] [backends and switches]',
                       #epilog="\n'.join(doc[1:])) # epilog not supported on my python2.4 machine: JDH")

    op.disable_interspersed_args()
    op.set_defaults(dirs='pylab,api,units,mplot3d',
                    clean=False, coverage=False, valgrind=False)
    op.add_option('-d', '--dirs', '--directories', type='string',
                  dest='dirs', help=dedent(''
Run only the tests in these directories; comma-separated list of
one or more of: pylab (or pylab_examples), api, units, mplot3d''))
    op.add_option('-b', '--backends', type='string', dest='backends',
                  help=dedent(''
Run tests only for these backends; comma-separated list of
one or more of: agg, ps, svg, pdf, template, cairo,
Default is everything except cairo.''))
    op.add_option('--clean', action='store_true', dest='clean',
                  help='Remove result directories, run no tests')
    op.add_option('--coverage', action='store_true', dest='coverage',
                  help='Run in coverage.py')
    op.add_option('--valgrind', action='store_true', dest='valgrind',
                  help='Run in valgrind')
options, args = op.parse_args()
switches = [x for x in args if x.startswith('--')]
backends = [x.lower() for x in args if not x.startswith('--')]
if options.backends:
    backends += [be.lower() for be in options.backends.split(',')]

result = Bunch(
    dirs = options.dirs.split(''),
    backends = backends or ['agg', 'ps', 'svg', 'pdf', 'template'],
    clean = options.clean,
    coverage = options.coverage,
    valgrind = options.valgrind,
    switches = switches)
if 'pylab_examples' in result.dirs:
    result.dirs[result.dirs.index('pylab_examples')] = 'pylab'
# print result
return (result)

if __name__ == '__main__':
times = {}
failures = {}
options = parse_options()
if options.clean:
    localdirs = [d for d in glob.glob('*') if os.path.isdir(d)]
    all_backends_set = set(all_backends)
    for d in localdirs:
        if d.lower() not in all_backends_set:
            continue
        print ('removing %s%d')
        for fname in glob.glob(os.path.join(d, '*')):
            os.remove(fname)
        os.rmdir(d)
    for fname in glob.glob('_tmp*.py'):
        os.remove(fname)

    print ('all clean...')
    raise SystemExit

if options.coverage:
    python = ['coverage.py', '-x']
elif options.valgrind:
    python = ['valgrind', '--tool=memcheck', '--leak-check=yes',
              '--log-file=%(name)s', sys.executable]
elif sys.platform == 'win32':
    python = [sys.executable]
else:
    python = [sys.executable]

report_all_missing(options.dirs)
for backend in options.backends:
    print ('testing %s %s' % (backend, ' '.join(options.switches)))
    t0 = time.time()
    failures[backend] = \

"""
drive(backend, options.dirs, python, options.switches)
t1 = time.time()
times[backend] = (t1-t0)/60.0

# print times
for backend, elapsed in times.items():
    print ('Backend %s took %1.2f minutes to complete' % (backend, elapsed))
    failed = failures[backend]
    if failed:
        print (' Failures: %d' % failed)
    if 'template' in times:
        print ('\tttemplate ratio %1.3f, template residual %1.3f' % (elapsed/times['template'], elapsed-times['template']))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
92.1 text_labels_and_annotations example code: rainbow_text.py

```python
# -*- coding: utf-8 -*-

""
The example shows how to string together several text objects.
""

HISTORY
-------
On the matplotlib-users list back in February 2012, Gökhan Sever asked the following question:
Is there a way in matplotlib to partially specify the color of a string?

Example:

```python
plt.ylabel("Today is cloudy.")
```

How can I show "today" as red, "is" as green and "cloudy." as blue?

Thanks.

Paul Ivanov responded with this answer:

```python
import matplotlib.pyplot as plt
from matplotlib.transforms import

def rainbow_text(x, y, strings, colors, ax=None, **kw):
    ""
    Take a list of `strings` and `colors` and place them next to each
    other, with text `strings[i]` being shown in `colors[i]`.

    This example shows how to do both vertical and horizontal text, and will
    pass all keyword arguments to plt.text, so you can set the font size,
    family, etc.

    The text will get added to the `ax` axes, if provided, otherwise the
    currently active axes will be used.
    ""
    if ax is None:
        ax = plt.gca()
    t = ax.transData
    canvas = ax.figure.canvas

    # horizontal version
    for s,c in zip(strings, colors):
        text = ax.text(x, y, " " + s + " ", color=c, transform=t, **kw)
        text.draw(canvas.get_renderer())
        ex = text.get_window_extent()
        t = transforms.offset_copy(text._transform, x=ex.width, units='dots')

    # vertical version
    for s,c in zip(strings, colors):
        text = ax.text(x, y, " " + s + " ", color=c, transform=t, 
                       rotation=90, va='bottom', ha='center', **kw)
        text.draw(canvas.get_renderer())
        ex = text.get_window_extent()
        t = transforms.offset_copy(text._transform, y=ex.height, units='dots')

    rainbow_text(0, 0, "all unicorns poop rainbows ! ! !".split(),
                 ["red", "cyan", "brown", "green", "blue", "purple", "black"],
                 size=18)

    plt.show()
```
92.2 text_labels_and_annotations example code: text_demo_fontdict.py

```python
import numpy as np
import matplotlib.pyplot as plt

font = {
    'family': 'serif',
    'color': 'darkred',
    'weight': 'normal',
    'size': 16,
}

x = np.linspace(0.0, 5.0, 100)
y = np.cos(2 * np.pi * x) * np.exp(-x)

plt.plot(x, y, 'k')
plt.title('Damped exponential decay', fontdict=font)
plt.text(2, 0.65, r'$\cos(2\pi t)\exp(-t)$', fontdict=font)
```

---

Demo using fontdict to control style of text and labels.

Damped exponential decay

$$\cos(2\pi t)\exp(-t)$$
plt.xlabel('time (s)', fontdict=font)
plt.ylabel('voltage (mV)', fontdict=font)

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.3 text_labels_and_annotations example code: unicode_demo.py

```
# -*- coding: utf-8 -*-

"""
Demo of unicode support in text and labels.
"""
from __future__ import unicode_literals

import matplotlib.pyplot as plt

plt.title('Développés et fabriqués')
plt.xlabel('réactivité nous permettent d'être sélectionnés et adoptés')
```
plt.ylabel('André was here!')
plt.text(0.2, 0.8, 'Institut für Festkörperphysik', rotation=45)
plt.text(0.4, 0.2, 'AVA (check kerning)')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Basic demo of axis spines.

This demo compares a normal axes, with spines on all four sides, and an axes with spines only on the left and bottom.

```python
import numpy as np
import matplotlib.pyplot as plt
```
x = np.linspace(0, 2 * np.pi, 100)
y = 2 * np.sin(x)

fig, (ax0, ax1) = plt.subplots(nrows=2)

ax0.plot(x, y)
ax0.set_title('normal spines')

ax1.plot(x, y)
ax1.set_title('bottom-left spines')

# Hide the right and top spines
ax1.spines['right'].set_visible(False)
ax1.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax1.yaxis.set_ticks_position('left')
ax1.xaxis.set_ticks_position('bottom')

# Tweak spacing between subplots to prevent labels from overlapping
plt.subplots_adjust(hspace=0.5)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of spines using custom bounds to limit the extent of the spine.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = y + 0.1 * np.random.normal(size=x.shape)

fig, ax = plt.subplots()
ax.plot(x, y, 'k--')
ax.plot(x, y2, 'ro')

# set ticks and tick labels
ax.set_xlim((0, 2*np.pi))
ax.set_xticks([0, np.pi, 2*np.pi])
ax.set_xticklabels([0, '$\pi$', '$2\pi$'])
ax.set_ylim([-1.5, 1.5])
ax.set_yticks([-1, 0, 1])
```

93.2 ticks_and_spines example code: spines_demo_bounds.py
import numpy as np
import matplotlib.pyplot as plt

# Only draw spine between the y-ticks
ax.spines['left'].set_bounds(-1, 1)
# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

plt.show()
fig, ax = plt.subplots()

image = np.random.uniform(size=(10, 10))
ax.imshow(image, cmap=plt.cm.gray, interpolation='nearest')
ax.set_title('dropped spines')

# Move left and bottom spines outward by 10 points
ax.spines['left'].set_position(('outward', 10))
ax.spines['bottom'].set_position(('outward', 10))

# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

93.4 ticks_and_spines example code: ticklabels_demo_rotation.py
Demo of custom tick-labels with user-defined rotation.

import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

plt.plot(x, y, 'ro')
plt.xticks(x, labels, rotation='vertical')
plt.margins(0.2)
plt.subplots_adjust(bottom=0.15)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
94.1 units example code: annotate_with_units.py

```python
import matplotlib.pyplot as plt
from basic_units import cm

fig, ax = plt.subplots()

ax.annotate( "Note 01", [0.5*cm, 0.5*cm] )

# xy and text both unitized
```
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
            xytext=(0.8*cm, 0.95*cm), textcoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

# mixing units w/ nonunits
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

ax.set_xlim(0*cm, 4*cm)
ax.set_ylim(0*cm, 4*cm)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

94.2 units example code: artist_tests.py

Test unit support with each of the matplotlib primitive artist types
The axes handles unit conversions and the artists keep a pointer to their axes parent, so you must init the artists with the axes instance if you want to initialize them with unit data, or else they will not know how to convert the units to scalars.

```python
import random
import matplotlib.lines as lines
import matplotlib.patches as patches
import matplotlib.text as text
import matplotlib.collections as collections

from basic_units import cm, inch
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.xaxis.set_units(cm)
ax.yaxis.set_units(cm)

if 0:
    # test a line collection
    # Not supported at present.
    verts = []
    for i in range(10):
        # a random line segment in inches
        verts.append(zip(*inch*10*np.random.rand(2, random.randint(2,15))))
    lc = collections.LineCollection(verts, axes=ax)
    ax.add_collection(lc)

    # test a plain-ol-line
    line = lines.Line2D([0*cm, 1.5*cm], [0*cm, 2.5*cm], lw=2, color='black', axes=ax)
    ax.add_line(line)

if 0:
    # test a patch
    # Not supported at present.
    rect = patches.Rectangle( (1*cm, 1*cm), width=5*cm, height=2*cm, alpha=0.2, axes=ax)
    ax.add_patch(rect)

    t = text.Text(3*cm, 2.5*cm, 'text label', ha='left', va='bottom', axes=ax)
    ax.add_artist(t)

    ax.set_xlim(-1*cm, 10*cm)
    ax.set_ylim(-1*cm, 10*cm)
    #ax.xaxis.set_units(inch)
    ax.grid(True)
    ax.set_title("Artists with units")
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
94.3 units example code: bar_demo2.py

```python
import numpy as np
from basic_units import cm, inch
import matplotlib.pyplot as plt

cms = cm * np.arange(0, 10, 2)
bottom=0*cm
width=0.8*cm

fig = plt.figure()

ax1 = fig.add_subplot(2,2,1)
ax1.bar(cms, cms, bottom=bottom)
```

plot using a variety of cm vs inches conversions. The example shows how default unit introspection works (ax1), how various keywords can be used to set the x and y units to override the defaults (ax2, ax3, ax4) and how one can set the xlimits using scalars (ax3, current units assumed) or units (conversions applied to get the numbers to current units)
ax2 = fig.add_subplot(2,2,2)
ax2.bar(cms, cms, bottom=bottom, width=width, xunits=cm, yunits=inch)

ax3 = fig.add_subplot(2,2,3)
ax3.bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=cm)
ax3.set_xlim(2, 6)  # scalars are interpreted in current units

ax4 = fig.add_subplot(2,2,4)
ax4.bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=inch)
#fig.savefig('simple_conversion_plot.png')
ax4.set_xlim(2*cm, 6*cm)  # cm are converted to inches

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

94.4 units example code: bar_unit_demo.py

```python
#!/usr/bin/env python
import numpy as np
from basic_units import cm, inch
import matplotlib.pyplot as plt
```

94.4. units example code: bar_unit_demo.py 2433
N = 5
menMeans = (150*cm, 160*cm, 146*cm, 172*cm, 155*cm)
menStd = (20*cm, 30*cm, 32*cm, 10*cm, 20*cm)

fig, ax = plt.subplots()

ind = np.arange(N) # the x locations for the groups
width = 0.35 # the width of the bars
p1 = ax.bar(ind, menMeans, width, color='r', bottom=0*cm, yerr=menStd)

womenMeans = (145*cm, 149*cm, 172*cm, 165*cm, 200*cm)
womenStd = (30*cm, 25*cm, 20*cm, 31*cm, 22*cm)
p2 = ax.bar(ind+width, womenMeans, width, color='y', bottom=0*cm, yerr=womenStd)

ax.set_title('Scores by group and gender')
ax.set_xticks(ind+width)
ax.set_xticklabels( ('G1', 'G2', 'G3', 'G4', 'G5') )

ax.legend( (p1[0], p2[0]), ('Men', 'Women') )
ax.yaxis.set_units(inch)
ax.autoscale_view()"
class TaggedValueMeta (type):
    def __init__(cls, name, bases, dict):
        for fn_name in cls._proxies.keys():
            try:
                dummy = getattr(cls, fn_name)
            except AttributeError:
                setattr(cls, fn_name,
                        ProxyDelegate(fn_name, cls._proxies[fn_name]))

class PassThroughProxy(object):
    def __init__(self, fn_name, obj):
        self.fn_name = fn_name
        self.target = obj.proxy_target

    def __call__(self, *args):
        fn = getattr(self.target, self.fn_name)
        ret = fn(*args)
        return ret

class ConvertArgsProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        converted_args = []
        for a in args:
            try:
                converted_args.append(a.convert_to(self.unit))
            except AttributeError:
                converted_args.append(TaggedValue(a, self.unit))
        converted_args = tuple([c.get_value() for c in converted_args])
        return PassThroughProxy.__call__(self, *converted_args)

class ConvertReturnProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        ret = PassThroughProxy.__call__(self, *args)
        if (type(ret) == type(NotImplemented)):
            return NotImplemented
        return TaggedValue(ret, self.unit)

class ConvertAllProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

94.5. units example code: basic_units.py
def __call__(self, *args):
    converted_args = []
    arg_units = [self.unit]
    for a in args:
        if hasattr(a, 'get_unit') and not hasattr(a, 'convert_to'):
            # if this arg has a unit type but no conversion ability,
            # this operation is prohibited
            return NotImplemented
        if hasattr(a, 'convert_to'):
            try:
                a = a.convert_to(self.unit)
            except:
                pass
            arg_units.append(a.get_unit())
            converted_args.append(a.get_value())
        else:
            converted_args.append(a)
            if hasattr(a, 'get_unit'):
                arg_units.append(a.get_unit())
            else:
                arg_units.append(None)
    converted_args = tuple(converted_args)
    ret = PassThroughProxy.__call__(self, *converted_args)
    if (type(ret) == type(NotImplemented)):
        return ret
    ret_unit = unit_resolver(self.fn_name, arg_units)
    if (ret_unit == NotImplemented):
        return ret
    return TaggedValue(ret, ret_unit)

class _TaggedValue(object):

    _proxies = {'__add__': ConvertAllProxy,
                '__sub__': ConvertAllProxy,
                '__mul__': ConvertAllProxy,
                '__rmul__': ConvertAllProxy,
                '__cmp__': ConvertAllProxy,
                '__lt__': ConvertAllProxy,
                '__gt__': ConvertAllProxy,
                '__len__': PassThroughProxy}

def __new__(cls, value, unit):
    # generate a new subclass for value
    value_class = type(value)
    try:
        subcls = type('TaggedValue_of_%s' % (value_class.__name__),
                      tuple([cls, value_class]),
                      {})
        if subcls not in units.registry:
            units.registry[subcls] = basicConverter
        return object.__new__(subcls, value, unit)
except TypeError:
    if cls not in units.registry:
        units.registry[cls] = basicConverter
        return object.__new__(cls, value, unit)

def __init__(self, value, unit):
    self.value = value
    self.unit = unit
    self.proxy_target = self.value

def __getattribute__(self, name):
    if name.startswith('__'):
        return object.__getattribute__(self, name)
    variable = object.__getattribute__(self, 'value')
    if (hasattr(variable, name) and name not in self.__class__.__dict__):
        return getattr(variable, name)
    return object.__getattribute__(self, name)

def __array__(self, t=None, context=None):
    if t is not None:
        return np.asarray(self.value).astype(t)
    else:
        return np.asarray(self.value, 'O')

def __array_wrap__(self, array, context):
    return TaggedValue(array, self.unit)

def __repr__(self):
    return 'TaggedValue(' + repr(self.value) + ', ' + repr(self.unit) + ')' 

def __str__(self):
    return str(self.value) + ' in ' + str(self.unit)

def __len__(self):
    return len(self.value)

def __iter__(self):
    class IteratorProxy(object):
        def __init__(self, iter, unit):
            self.iter = iter
            self.unit = unit

        def __next__(self):
            value = next(self.iter)
            return TaggedValue(value, self.unit)
        next = __next__  # for Python 2
        __next__ = next

    return IteratorProxy(iter(self.value), self.unit)

def get_compressed_copy(self, mask):
    new_value = np.ma.masked_array(self.value, mask=mask).compressed()
    return TaggedValue(new_value, self.unit)

def convert_to(self, unit):

94.5. units example code: basic_units.py
if (unit == self.unit or not unit):
    return self
new_value = self.unit.convert_value_to(self.value, unit)
return TaggedValue(new_value, unit)

def get_value(self):
    return self.value

def get_unit(self):
    return self.unit

TaggedValue = TaggedValueMeta('TaggedValue', (TaggedValue, ), {})

class BasicUnit(object):
    def __init__(self, name, fullname=None):
        self.name = name
        if fullname is None:
            fullname = name
        self.fullname = fullname
        self.conversions = dict()

    def __repr__(self):
        return 'BasicUnit(%s)'%self.name

    def __str__(self):
        return self.fullname

    def __call__(self, value):
        return TaggedValue(value, self)

    def __mul__(self, rhs):
        value = rhs
        unit = self
        if hasattr(rhs, 'get_unit'):
            value = rhs.get_value()
            unit = rhs.get_unit()
            unit = unit_resolver('__mul__', (self, unit))
        if (unit == NotImplemented):
            return NotImplemented
        return TaggedValue(value, unit)

    def __rmul__(self, lhs):
        return self*lhs

    def __array_wrap__(self, array, context):
        return TaggedValue(array, self)

    def __array__(self, t=None, context=None):
        ret = np.array([1])
        if t is not None:
            return ret.astype(t)
else:
    return ret

def add_conversion_factor(self, unit, factor):
    def convert(x):
        return x*factor
    self.conversions[unit] = convert

def add_conversion_fn(self, unit, fn):
    self.conversions[unit] = fn

def get_conversion_fn(self, unit):
    return self.conversions[unit]

def convert_value_to(self, value, unit):
    conversion_fn = self.conversions[unit]
    ret = conversion_fn(value)
    return ret

def get_unit(self):
    return self

class UnitResolver(object):
    def addition_rule(self, units):
        for unit_1, unit_2 in zip(units[:-1], units[1:]):
            if (unit_1 != unit_2):
                return NotImplemented
        return units[0]

    def multiplication_rule(self, units):
        non_null = [u for u in units if u]
        if (len(non_null) > 1):
            return NotImplemented
        return non_null[0]

    op_dict = {
        '__mul__': multiplication_rule,
        '__rmul__': multiplication_rule,
        '__add__': addition_rule,
        '__radd__': addition_rule,
        '__sub__': addition_rule,
        '__rsub__': addition_rule
    }

    def __call__(self, operation, units):
        if (operation not in self.op_dict):
            return NotImplemented
        return self.op_dict[operation](self, units)

unit Resolver = UnitResolver()
cm = BasicUnit('cm', 'centimeters')
in = BasicUnit('inch', 'inches')
in.add_conversion_factor(cm, 2.54)
cm.add_conversion_factor(in, 1/2.54)

radians = BasicUnit('rad', 'radians')
degrees = BasicUnit('deg', 'degrees')
radians.add_conversion_factor(degrees, 180.0/np.pi)
degrees.add_conversion_factor(radians, np.pi/180.0)

secs = BasicUnit('s', 'seconds')
hertz = BasicUnit('Hz', 'Hertz')
minutes = BasicUnit('min', 'minutes')

secs.add_conversion_fn(hertz, lambda x: 1./x)
secs.add_conversion_factor(minutes, 1/60.0)

# radians formatting
def rad_fn(x, pos=None):
    n = int((x / np.pi) * 2.0 + 0.25)
    if n == 0:
        return '0'
    elif n == 1:
        return r'$\pi/2$'
    elif n == 2:
        return r'$\pi$'
    elif n % 2 == 0:
        return r'$%i\pi$' % (n/2,)
    else:
        return r'$%i\pi/2$' % (n,)

class BasicUnitConverter(units.ConversionInterface):
    @staticmethod
    def axisinfo(unit, axis):
        'Return AxisInfo instance for x and unit'
        if unit==radians:
            return units.AxisInfo(
                majloc=ticker.MultipleLocator(base=np.pi/2),
                majfmt=ticker.FuncFormatter(rad_fn),
                label=unit.fullname,
            )
        elif unit==degrees:
            return units.AxisInfo(
                majloc=ticker.AutoLocator(),
                majfmt=ticker.FormatStrFormatter(r'$%i^\circ$'),
                label=unit.fullname,
            )
        elif unit is not None:
            if hasattr(unit, 'fullname'):
```
return units.AxisInfo(label=unit.fullname)
elif hasattr(unit, 'unit'):
    return units.AxisInfo(label=unit.unit.fullname)
return None

@staticmethod
def convert(val, unit, axis):
    if units.ConversionInterface.is_numlike(val):
        return val
    if iterable(val):
        return [thisval.convert_to(unit).get_value() for thisval in val]
    else:
        return val.convert_to(unit).get_value()

@staticmethod
def default_units(x, axis):
    'return the default unit for x or None'
    if iterable(x):
        for thisx in x:
            return thisx.unit
    return x.unit

def cos(x):
    if iterable(x):
        return [math.cos(val.convert_to(radians).get_value()) for val in x]
    else:
        return math.cos(x.convert_to(radians).get_value())

basicConverter = BasicUnitConverter()
units.registry[BasicUnit] = basicConverter
units.registry[TaggedValue] = basicConverter
```

---

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
94.6 units example code: ellipse_with_units.py
Compare the ellipse generated with arcs versus a polygonal approximation

```python
from basic_units import cm
import numpy as np
import matplotlib.patches
import matplotlib.pyplot as plt

xcenter, ycenter = 0.38*cm, 0.52*cm
#xcenter, ycenter = 0., 0.
width, height = 1e-1*cm, 3e-1*cm
angle = -30

theta = np.arange(0.0, 360.0, 1.0)*np.pi/180.0
x = 0.5*width*np.cos(theta)
y = 0.5*height*np.sin(theta)

rtheta = angle*np.pi/180.
R = np.array([[np.cos(rtheta), -np.sin(rtheta)],
              [np.sin(rtheta), np.cos(rtheta)]])
```

94.6. units example code: ellipse_with_units.py
x, y = np.dot(R, np.array([x, y]))
x += xcenter
y += ycenter

fig = plt.figure()
ax = fig.add_subplot(211, aspect="auto")
a.fill(x, y, alpha=0.2, facecolor='yellow', edgecolor='yellow', linewidth=1, zorder=1)

e1 = patches.Ellipse((xcenter, ycenter), width, height,
  angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e1)

ax = fig.add_subplot(212, aspect="equal")
a.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Ellipse((xcenter, ycenter), width, height,
  angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e2)

#fig.savefig('ellipse_compare.png')
fig.savefig('ellipse_compare')

fig = plt.figure()
ax = fig.add_subplot(211, aspect="auto")
a.fill(x, y, alpha=0.2, facecolor='yellow', edgecolor='yellow', linewidth=1, zorder=1)

e1 = patches.Arc((xcenter, ycenter), width, height,
  angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e1)

ax = fig.add_subplot(212, aspect="equal")
a.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Arc((xcenter, ycenter), width, height,
  angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e2)

#fig.savefig('arc_compare.png')
fig.savefig('arc_compare')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
A mockup "Foo" units class which supports conversion and different tick formatting depending on the "unit". Here the "unit" is just a scalar conversion factor, but this example shows mpl is entirely agnostic to what kind of units client packages use

```python
from matplotlib.cbook import iterable
import matplotlib.units as units
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt

class Foo:
    def __init__(self, val, unit=1.0):
        self.unit = unit
        self._val = val * unit

    def value(self, unit):
        if unit is None: unit = self.unit
        return self._val / unit

class FooConverter:
```

---

94.7 units example code: evans_test.py
@staticmethod
def axisinfo(unit, axis):
    'return the Foo AxisInfo'
    if unit==1.0 or unit==2.0:
        return units.AxisInfo(
            majloc = ticker.IndexLocator( 8, 0 ),
            majfmt = ticker.FormatStrFormatter("VAL: %s"),
            label='foo',
        )
    else:
        return None

@staticmethod
def convert(obj, unit, axis):
    ""
    convert obj using unit. If obj is a sequence, return the
    converted sequence
    ""
    if units.ConversionInterface.is_numlike(obj):
        return obj
    if iterable(obj):
        return [o.value(unit) for o in obj]
    else:
        return obj.value(unit)

@staticmethod
def default_units(x, axis):
    'return the default unit for x or None'
    if iterable(x):
        for thisx in x:
            return thisx.unit
    else:
        return x.unit

units.registry[Foo] = FooConverter()

# create some Foos
x = []
for val in range( 0, 50, 2 ):
    x.append( Foo( val, 1.0 ) )

# and some arbitrary y data
y = [i for i in range( len(x) )]

# plot specifying units
fig = plt.figure()
fig.suptitle("Custom units")
fig.subplots_adjust(bottom=0.2)
al = fig.add_subplot(1,2,2)
ax.plot( x, y, 'o', xunits=2.0 )
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')
ax.set_title("\textit{xunits} = 2.0")

# plot without specifying units; will use the None branch for axisinfo
ax = fig.add_subplot(1,2,1)
ax.plot(x, y)  # uses default units
ax.set_title("default units")
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

94.8 units example code: radian_demo.py

---

Plot with radians from the basic_units mockup example package
This example shows how the unit class can determine the tick locating, formatting and axis labeling.

```
import numpy as np
from basic_units import radians, degrees, cos
from matplotlib.pyplot import figure, show

x = [val*radians for val in np.arange(0, 15, 0.01)]

fig = figure()
fig.subplots_adjust(hspace=0.3)

ax = fig.add_subplot(211)
line1, = ax.plot(x, cos(x), xunits=radians)

ax = fig.add_subplot(212)
line2, = ax.plot(x, cos(x), xunits=degrees)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
plot using a variety of cm vs inches conversions. The example shows how default unit introspection works (ax1), how various keywords can be used to set the x and y units to override the defaults (ax2, ax3, ax4) and how one can set the xlimits using scalars (ax3, current units assumed) or units (conversions applied to get the numbers to current units)

```
from basic_units import cm, inch
import matplotlib.pyplot as plt
import numpy

cms = cm *numpy.arange(0, 10, 2)

fig = plt.figure()

ax1 = fig.add_subplot(2,2,1)
ax1.plot(cms, cms)

ax2 = fig.add_subplot(2,2,2)
ax2.plot(cms, cms, xunits=cm, yunits=inch)

ax3 = fig.add_subplot(2,2,3)
ax3.plot(cms, cms, xunits=inch, yunits=cm)
ax3.set_xlim(3, 6)  # scalars are interpreted in current units

ax4 = fig.add_subplot(2,2,4)
ax4.plot(cms, cms, xunits=inch, yunits=inch)
#fig.savefig('simple_conversion_plot.png')
ax4.set_xlim(3*cm, 6*cm)  # cm are converted to inches

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
94.10 units example code: units_scatter.py

```python
""
Demonstrate unit handling

basic_units is a mockup of a true units package used for testing purposes, which illustrates the basic interface that a units package must provide to matplotlib.

The example below shows support for unit conversions over masked arrays.
""
import numpy as np
from basic_units import secs, hertz, minutes
from matplotlib.pylab import figure, show

# create masked array

xsecs = secs*np.ma.MaskedArray((1,2,3,4,5,6,7,8), (1,0,1,0,0,0,1,0), np.float)
#xsecs = secs*np.arange(1,10.)
fig = figure()
ax1 = fig.add_subplot(3,1,1)
```

The code above demonstrates the usage of units with masked arrays in Matplotlib. It imports necessary packages, defines a masked array, and plots the data with different units (seconds, hertz, minutes). The example shows how to handle units with masked arrays in Matplotlib.
ax1.scatter(xsecs, xsecs)
#ax1.set_ylabel('seconds')
ax1.axis([0,10,0,10])

ax2 = fig.add_subplot(3,1,2, sharex=ax1)
ax2.scatter(xsecs, xsecs, yunits=hertz)
ax2.axis([0,10,0,1])

ax3 = fig.add_subplot(3,1,3, sharex=ax1)
ax3.scatter(xsecs, xsecs, yunits=hertz)
ax3.yaxis.set_units(minutes)
ax3.axis([0,10,0,1])

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
95.1 user_interfaces example code: embedding_in_gtk.py

[source code]

#!/usr/bin/env python

"""
show how to add a matplotlib FigureCanvasGTK or FigureCanvasGTKAgg widget to a
gtk.Window
"""

import gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi

# uncomment to select /GTK/GTKAgg/GTKCairo
#from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
#from matplotlib.backends.backend_gtkcairo import FigureCanvasGTKCairo as FigureCanvas

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0,3.0,0.01)
s = sin(2*pi*t)
a.plot(t,s)

canvas = FigureCanvas(f)  # a gtk.DrawingArea
win.add(canvas)

win.show_all()
gtk.main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
95.2 user_interfaces example code: embedding_in_gtk2.py

[source code]

#!/usr/bin/env python

"""show how to add a matplotlib FigureCanvasGTK or FigureCanvasGTKAgg widget and a toolbar to a gtk.Window
"""

import gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi

# uncomment to select /GTK/GTKAgg/GTKCairo
#from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
#or NavigationToolbar for classic
#from matplotlib.backends.backend_gtk import NavigationToolbar2GTK as NavigationToolbar
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as NavigationToolbar

# implement the default mpl key bindings
from matplotlib.backend_bases import key_press_handler

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

vbox = gtk.VBox()
win.add(vbox)

fig = Figure(figsize=(5,4), dpi=100)
ax = fig.add_subplot(111)
t = arange(0.0,3.0,0.01)
s = sin(2*pi*t)

ax.plot(t,s)

canvas = FigureCanvas(fig) # a gtk.DrawingArea
vbox.pack_start(canvas)
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False)

def on_key_event(event):
    print('you pressed %s' % event.key)
    key_press_handler(event, canvas, toolbar)

canvas.mpl_connect('key_press_event', on_key_event)
```python
#!/usr/bin/env python

"""
demonstrate adding a FigureCanvasGTK3Agg widget to a Gtk.ScrolledWindow
using GTK3 accessed via pygobject
"""

from gi.repository import Gtk
from matplotlib.figure import Figure
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0,3.0,0.01)
s = sin(2*pi*t)
a.plot(t,s)

sw = Gtk.ScrolledWindow()
win.add(sw)
# A scrolled window border goes outside the scrollbars and viewport
sw.set_border_width(10)

canvas = FigureCanvas(f)  # a Gtk.DrawingArea
canvas.set_size_request(800,600)
sw.add_with_viewport (canvas)

win.show_all()
Gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
#!/usr/bin/env python

"""
demonstrate NavigationToolbar with GTK3 accessed via pygobject
"""

from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3 import NavigationToolbar2GTK3 as NavigationToolbar

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(1,1,1)
t = arange(0.0,3.0,0.01)
s = sin(2*pi*t)
a.plot(t,s)

vbox = Gtk.VBox()
win.add(vbox)

# Add canvas to vbox
canvas = FigureCanvas(f)  # a Gtk.DrawingArea
vbox.pack_start(canvas, True, True, 0)

# Create toolbar
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False, 0)

win.show_all()
Gtk.main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.5 user_interfaces example code: embedding_in_qt4.py

[source code]

#!/usr/bin/env python

# embedding_in_qt4.py --- Simple Qt4 application embedding matplotlib canvases
#
# Copyright (C) 2005 Florent Rougon
#                2006 Darren Dale
#
# This file is an example program for matplotlib. It may be used and
# modified with no restriction; raw copies as well as modified versions
# may be distributed without limitation.

from __future__ import unicode_literals
import sys
import os
import random
from matplotlib.backends import qt4_compat
use_pyside = qt4_compat.QT_API == qt4_compat.QT_API_PYSIDE
if use_pyside:
    from PySide import QtGui, QtCore
else:
    from PyQt4 import QtGui, QtCore

from numpy import arange, sin, pi
from matplotlib.backends.backend_qt4agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure

progname = os.path.basename(sys.argv[0])
progversion = "0.1"

class MyMplCanvas(FigureCanvas):
    """Ultimately, this is a QWidget (as well as a FigureCanvasAgg, etc.).""
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        # We want the axes cleared every time plot() is called
        self.axes.hold(False)

        self.compute_initial_figure()

    #
    FigureCanvas.__init__(self, fig)
    self.setParent(parent)

    FigureCanvas.setSizePolicy(self,
                                QtGui.QSizePolicy.Expanding,
                                QtGui.QSizePolicy.Expanding)
    FigureCanvas.updateGeometry(self)

    def compute_initial_figure(self):
        pass

class MyStaticMplCanvas(MyMplCanvas):
    """Simple canvas with a sine plot.""
    def compute_initial_figure(self):
        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)
        self.axes.plot(t, s)
class MyDynamicMplCanvas(MyMplCanvas):
    """A canvas that updates itself every second with a new plot."""
    def __init__(self, *args, **kwargs):
        MyMplCanvas.__init__(self, *args, **kwargs)
        timer = QtCore.QTimer(self)
        timer.timeout.connect(self.update_figure)
        timer.start(1000)

    def compute_initial_figure(self):
        self.axes.plot([0, 1, 2, 3], [1, 2, 0, 4], 'r')

    def update_figure(self):
        # Build a list of 4 random integers between 0 and 10 (both inclusive)
        l = [random.randint(0, 10) for i in range(4)]

        self.axes.plot([0, 1, 2, 3], l, 'r')
        self.draw()

class ApplicationWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setAttribute(QtCore.Qt.WA_DeleteOnClose)
        self.setWindowTitle("application main window")

        self.file_menu = QtGui.QMenu(
            '&File', self)
        self.file_menu.addAction(
            '&Quit', self.fileQuit,
            QtCore.Qt.CTRL + QtCore.Qt.Key_Q)
        self.menuBar().addMenu(self.file_menu)

        self.help_menu = QtGui.QMenu(
            '&Help', self)
        self.menuBar().addSeparator()
        self.menuBar().addMenu(self.help_menu)

        self.help_menu.addAction(
            '&About', self.about)

        self.main_widget = QtGui.QWidget(self)

        l = QtGui.QVBoxLayout(self.main_widget)
        sc = MyStaticMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        dc = MyDynamicMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        l.addWidget(sc)
        l.addWidget(dc)

        self.main_widget.setFocus()
        self.setCentralWidget(self.main_widget)
        self.statusBar().showMessage("All hail matplotlib!", 2000)

    def fileQuit(self):
        self.close()

    def closeEvent(self, ce):
        pass
self.fileQuit()

def about(self):
    QtGui.QMessageBox.about(self, "About",
    """embedding_in_qt4.py example
Copyright 2005 Florent Rougon, 2006 Darren Dale

This program is a simple example of a Qt4 application embedding matplotlib canvases.

It may be used and modified with no restriction; raw copies as well as modified versions may be distributed without limitation.""
    )

qApp = QtGui.QApplication(sys.argv)
aw = ApplicationWindow()
aw.setWindowTitle("%s" % proname)
aw.show()
sys.exit(qApp.exec_())
#qApp.exec_()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.6 user_interfaces example code: embedding_in_qt4_wtoolbar.py

[source code]

from __future__ import print_function

import sys

import numpy as np
from matplotlib.figure import Figure
from matplotlib.backend_bases import key_press_handler
from matplotlib.backends.backend_qt4agg import (FigureCanvasQTAgg as FigureCanvas,
NavigationToolbar2QT as NavigationToolbar)

from matplotlib.backends import qt4_compat
use_pyside = qt4_compat.QT_API == qt4_compat.QT_API_PYSIDE

if use_pyside:
    from PySide.QtCore import *
    from PySide.QtGui import *
else:
    from PyQt4.QtCore import *
    from PyQt4.QtGui import *

class AppForm(QMainWindow):

95.6. user_interfaces example code: embedding_in_qt4_wtoolbar.py 2459
def __init__(self, parent=None):
    QMainWindow.__init__(self, parent)
    #self.x, self.y = self.get_data()
    self.data = self.get_data2()
    self.create_main_frame()
    self.on_draw()

def create_main_frame(self):
    self.main_frame = QWidget()
    self.fig = Figure((5.0, 4.0), dpi=100)
    self.canvas = FigureCanvas(self.fig)
    self.canvas.setParent(self.main_frame)
    self.canvas.setFocusPolicy(Qt.StrongFocus)
    self.canvas.setFocus()
    self.mpl_toolbar = NavigationToolbar(self.canvas, self.main_frame)
    self.canvas.mpl_connect('key_press_event', self.on_key_press)

    vbox = QVBoxLayout()
    vbox.addWidget(self.canvas)
    vbox.addWidget(self.mpl_toolbar)
    self.main_frame.setLayout(vbox)
    self.setCentralWidget(self.main_frame)

def get_data2(self):
    return np.arange(20).reshape([4, 5]).copy()

def on_draw(self):
    self.fig.clear()
    self.axes = self.fig.add_subplot(111)
    #self.axes.plot(self.x, self.y, 'ro')
    self.axes.imshow(self.data, interpolation='nearest')
    #self.axes.plot([1,2,3])
    self.canvas.draw()

def on_key_press(self, event):
    print('you pressed', event.key)
    # implement the default mpl key press events described at
    # http://matplotlib.org/users/navigation_toolbar.html#navigation-keyboard-shortcuts
    key_press_handler(event, self.canvas, self.mpl_toolbar)

def main():
    app = QApplication(sys.argv)
    form = AppForm()
    form.show()
    app.exec_()

if __name__ == '__main__':
    main()
95.7 user_interfaces example code: embedding_in_tk.py

[source code]

#!/usr/bin/env python

import matplotlib
matplotlib.use('TkAgg')

from numpy import arange, sin, pi
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
# implement the default mpl key bindings
from matplotlib.backends.backend_bases import key_press_handler

from matplotlib.figure import Figure
import sys
if sys.version_info[0] < 3:
    import Tkinter as Tk
else:
    import tkinter as Tk

root = Tk.Tk()
root.wm_title("Embedding in TK")

f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0,3.0,0.01)
s = sin(2*pi*t)
a.plot(t,s)

# a tk.DrawingArea
canvas = FigureCanvasTkAgg(f, master=root)
canvas.show()
canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

toolbar = NavigationToolbar2TkAgg( canvas, root )
toolbar.pack(side=Tk.TOP)
toolbar.update()
canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

def on_key_event(event):
    print('you pressed %s'%event.key)
    key_press_handler(event, canvas, toolbar)

canvas.mpl_connect('key_press_event', on_key_event)
```python
def _quit():
    root.quit()  # stops mainloop
    root.destroy()  # this is necessary on Windows to prevent
    # Fatal Python Error: PyEval_RestoreThread: NULL tstate

button = Tk.Button(master=root, text='Quit', command=_quit)
button.pack(side=Tk.BOTTOM)

Tk.mainloop()

# If you put root.destroy() here, it will cause an error if
# the window is closed with the window manager.

Keywords: python, matplotlib, p自理, example, codex (see Search examples)

95.8 user_interfaces example code: embedding_in_tk2.py

[source code]

#!/usr/bin/env python
import matplotlib
matplotlib.use('TkAgg')

from numpy import arange, sin, pi
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
from matplotlib.figure import Figure

import sys
if sys.version_info[0] < 3:
    import Tkinter as Tk
else:
    import tkinter as Tk

def destroy(e): sys.exit()

root = Tk.Tk()
root.wm_title("Embedding in TK")
#root.bind("<Destroy>", destroy)

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)
a.set_title('Tk embedding')
a.set_xlabel('X axis label')
a.set_ylabel('Y label')
```
# a tk.DrawingArea
canvas = FigureCanvasTkAgg(f, master=root)
canvas.show()
canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

#toolbar = NavigationToolbar2TkAgg( canvas, root )
#toolbar.update()
canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

button = Tk.Button(master=root, text='Quit', command=sys.exit)
button.pack(side=Tk.BOTTOM)

Tk.mainloop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.9 user_interfaces example code: embedding_in_wx2.py

[source code]

#!/usr/bin/env python
'''
An example of how to use wx or wxagg in an application with the new toolbar - comment out the setA_toolbar line for no toolbar
'''

# Used to guarantee to use at least Wx2.8
import wxversion
wxversion.ensureMinimal('2.8')

from numpy import arange, sin, pi

import matplotlib

# uncomment the following to use wx rather than wxagg
#matplotlib.use('WX')
#from matplotlib.backends.backend_wx import FigureCanvasWx as FigureCanvas

# comment out the following to use wx rather than wxagg
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

from matplotlib.backends.backend_wx import NavigationToolbar2Wx

from matplotlib.figure import Figure

import wx

class CanvasFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1,
    'CanvasFrame', size=(550, 350))

self.SetBackgroundColour(wx.NamedColour("WHITE"))

self.figure = Figure()
self.axes = self.figure.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)

self.axes.plot(t, s)
self.canvas = FigureCanvas(self, -1, self.figure)

self.sizer = wx.BoxSizer(wx.VERTICAL)
self.sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
self.SetSizer(self.sizer)
self.Fit()

self.add_toolbar()  # comment this out for no toolbar

def add_toolbar(self):
    self.toolbar = NavigationToolbar2Wx(self.canvas)
    self.toolbar.Realize()
    if wx.Platform == '__WXMAC__':
        # Mac platform (OSX 10.3, MacPython) does not seem to cope with
        # having a toolbar in a sizer. This work-around gets the buttons
        # back, but at the expense of having the toolbar at the top
        self.SetToolBar(self.toolbar)
    else:
        # On Windows platform, default window size is incorrect, so set
        # toolbar width to figure width.
        tw, th = self.toolbar.GetSizeTuple()
        fw, fh = self.canvas.GetSizeTuple()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        # As noted above, doesn't work for Mac.
        self.toolbar.SetSize(wx.Size(fw, th))
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        # update the axes menu on the toolbar
        self.toolbar.update()

def OnPaint(self, event):
    self.canvas.draw()

class App(wx.App):
    def OnInit(self):
        'Create the main window and insert the custom frame'
        frame = CanvasFrame()
        frame.Show(True)
`return True`

```python
app = App(0)
app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 95.10 user_interfaces example code: embedding_in_wx3.py

[source code]

```python
#!/usr/bin/env python

""
Copyright (C) 2003-2004 Andrew Straw, Jeremy ODonoghue and others

License: This work is licensed under the PSF. A copy should be included with this source code, and is also available at http://www.python.org/psf/license.html

This is yet another example of using matplotlib with wx. Hopefully this is pretty full-featured:

- both matplotlib toolbar and WX buttons manipulate plot
- full wxApp framework, including widget interaction
- XRC (XML wxWidgets resource) file to create GUI (made with XRCed)

This was derived from embedding_in_wx and dynamic_image_wxagg.

Thanks to matplotlib and wx teams for creating such great software!

""

```python
from __future__ import print_function

# Used to guarantee to use at least Wx2.8
import wxversion
wxversion.ensureMinimal('2.8')

import sys, time, os, gc
import matplotlib
matplotlib.use('WXAgg')
import matplotlib.cm as cm
import matplotlib.cbook as cbook
from matplotlib.backends.backend_wxagg import Toolbar, FigureCanvasWxAgg
from matplotlib.figure import Figure
import numpy as np

import wx
import wx.xrc as xrc

ERR_TOL = 1e-5 # floating point slop for peak-detection
```
matplotlib.rc('image', origin='lower')

class PlotPanel(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent, -1)
        self.fig = Figure((5,4), 75)
        self.canvas = FigureCanvasWxAgg(self, -1, self.fig)
        self.toolbar = Toolbar(self.canvas)  #matplotlib toolbar
        self.toolbar.Realize()
        #self.toolbar.set_active([0,1])
        # Now put all into a sizer
        sizer = wx.BoxSizer(wx.VERTICAL)
        # This way of adding to sizer allows resizing
        sizer.Add(self.canvas, 1, wx.LEFT|wx.TOP|wx.GROW)
        # Best to allow the toolbar to resize!
        sizer.Add(self.toolbar, 0, wx.GROW)
        self.SetSizer(sizer)
        self.Fit()

    def init_plot_data(self):
        a = self.fig.add_subplot(111)
        x = np.arange(120.0)*2*np.pi/60.0
        y = np.arange(100.0)*2*np.pi/50.0
        self.x, self.y = np.meshgrid(x, y)
        z = np.sin(self.x) + np.cos(self.y)
        self.im = a.imshow(z, cmap=cm.jet)#, interpolation='nearest')
        zmax = np.amax(z) - ERR_TOL
        ymax_i, xmax_i = np.nonzero(z >= zmax)
        if self.im.origin == 'upper':
            ymax_i = z.shape[0]-ymax_i
        self.lines = a.plot(xmax_i,ymax_i,
                           'ko')
        self.toolbar.update()  # Not sure why this is needed - ADS

    def GetToolBar(self):
        # You will need to override GetToolBar if you are using an
        # unmanaged toolbar in your frame
        return self.toolbar

    def OnWhiz(self,evt):
        self.x += np.pi/15
        self.y += np.pi/20
        z = np.sin(self.x) + np.cos(self.y)
        self.im.set_array(z)
        zmax = np.amax(z) - ERR_TOL
        ymax_i, xmax_i = np.nonzero(z >= zmax)
if self.im.origin == 'upper':
    ymax_i = z.shape[0] - ymax_i
    self.lines[0].set_data(xmax_i, ymax_i)
    self.canvas.draw()

def onEraseBackground(self, evt):
    # this is supposed to prevent redraw flicker on some X servers...
    pass

class MyApp(wx.App):
    def OnInit(self):
        xrcfile = cbook.get_sample_data('embedding_in_wx3.xrc', asfileobj=False)
        print('loading', xrcfile)

        self.res = xrc.XmlResource(xrcfile)

        # main frame and panel --------
        self.frame = self.res.LoadFrame(None, 'MainFrame')
        self.panel = xrc.XRCCTRL(self.frame, 'MainPanel')

        # matplotlib panel ------------
        # container for matplotlib panel (I like to make a container
        # panel for our panel so I know where it'll go when in XRced.)
        plot_container = xrc.XRCCTRL(self.frame, 'plot_container_panel')
        sizer = wx.BoxSizer(wx.VERTICAL)

        # matplotlib panel itself
        self.plotpanel = PlotPanel(plot_container)
        self.plotpanel.init_plot_data()

        # wx boilerplate
        sizer.Add(self.plotpanel, 1, wx.EXPAND)
        plot_container.SetSizer(sizer)

        # whiz button ----------------
        whiz_button = xrc.XRCCTRL(self.frame, 'whiz_button')
        wx.EVT_BUTTON(whiz_button, whiz_button.GetId(),
                      self.plotpanel.OnWhiz)

        # bang button ---------------
        bang_button = xrc.XRCCTRL(self.frame, 'bang_button')
        wx.EVT_BUTTON(bang_button, bang_button.GetId(),
                      self.OnBang)

        # final setup ---------------
        sizer = self.panel.GetSizer()
        self.frame.Show(1)
self.SetTopWindow(self.frame)

return True

def OnBang(self, event):
    bang_count = xrc.XRCCTRL(self.frame, "bang_count")
    bangs = bang_count.GetValue()
    bangs = int(bangs) + 1
    bang_count.SetValue(str(bangs))

if __name__ == '__main__':
    app = MyApp(0)
    app.MainLoop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.11 user_interfaces example code: embedding_in_wx4.py

[source code]

#!/usr/bin/env python

""
An example of how to use wx or wxagg in an application with a custom toolbar
""

# Used to guarantee to use at least Wx2.8
import wxversion
wxversion.ensureMinimal('2.8')

from numpy import arange, sin, pi

import matplotlib

matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wxagg import NavigationToolbar2WxAgg

from matplotlib.backends.backend_wx import _load_bitmap
from matplotlib.figure import Figure
from numpy.random import rand

import wx

class MyNavigationToolbar(NavigationToolbar2WxAgg):
    """
    Extend the default wx toolbar with your own event handlers
    """
    ON_CUSTOM = wx.NewId()
    def __init__(self, canvas, cankill):
        super(MyNavigationToolbar, self).

Chapter 95. user_interfaces Examples
NavigationToolbar2WxAgg.__init__(self, canvas)

# for simplicity I'm going to reuse a bitmap from wx, you'll
# probably want to add your own.
self.AddSimpleTool(self.ON_CUSTOM, _load_bitmap('stock_left.xpm'),
      'Click me', 'Activate custom control')
wx.EVT_TOOL(self, self.ON_CUSTOM, self._on_custom)

def _on_custom(self, evt):
    # add some text to the axes in a random location in axes (0,1)
    # coords) with a random color

    # get the axes
    ax = self.canvas.figure.axes[0]

    # generate a random location can color
    x, y = tuple(rand(2))
    rgb = tuple(rand(3))

    # add the text and draw
    ax.text(x, y, 'You clicked me',
            transform=ax.transAxes,
            color=rgb)
    self.canvas.draw()
    evt.Skip()

class CanvasFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self,None,-1,
                          'CanvasFrame',size=(550,350))
        self.SetBackgroundColour(wx.NamedColour("WHITE"))

        self.figure = Figure(figsize=(5,4), dpi=100)
        self.axes = self.figure.add_subplot(111)
        t = arange(0.0,3.0,0.01)
        s = sin(2*pi*t)

        self.axes.plot(t,s)

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.TOP | wx.LEFT | wx.EXPAND)
        # Capture the paint message
        wx.EVT_PAINT(self, self.OnPaint)

        self.toolbar = MyNavigationToolbar(self.canvas, True)
        self.toolbar.Realize()

        if wx.Platform == '__WXMAC__':
            # Mac platform (OSX 10.3, MacPython) does not seem to cope with

95.11. user_interfaces example code: embedding_in_wx4.py
# having a toolbar in a sizer. This work-around gets the buttons back, but at the expense of having the toolbar at the top
self.SetToolBar(self.toolbar)
else:
    # On Windows platform, default window size is incorrect, so set toolbar width to figure width.
    tw, th = self.toolbar.GetSizeTuple()
    fw, fh = self.canvas.GetSizeTuple()
    # By adding toolbar in sizer, we are able to put it at the bottom of the frame - so appearance is closer to GTK version.
    # As noted above, doesn't work for Mac.
    self.toolbar.SetSize(wx.Size(fw, th))
    self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)

    # update the axes menu on the toolbar
    self.toolbar.update()
    self.SetSizer(self.sizer)
    self.Fit()

def OnPaint(self, event):
    self.canvas.draw()
    event.Skip()

class App(wx.App):
    def OnInit(self):
        'Create the main window and insert the custom frame'
        frame = CanvasFrame()
        frame.Show(True)

        return True

app = App(0)
app.MainLoop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.12 user_interfaces example code: embedding_in_wx5.py

[source code]

# Used to guarantee to use at least Wx2.8
import wxversion
wxversion.ensureMinimal('2.8')

import wx
import wx.aui
import matplotlib as mpl
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as Canvas
from matplotlib.backends.backend_wxagg import NavigationToolbar2Wx as Toolbar
class Plot(wx.Panel):
    def __init__(self, parent, id = -1, dpi = None, **kwargs):
        wx.Panel.__init__(self, parent, id=id, **kwargs)
        self.figure = mpl.figure.Figure(dpi=dpi, figsize=(2,2))
        self.canvas = Canvas(self, -1, self.figure)
        self.toolbar = Toolbar(self.canvas)
        self.toolbar.Realize()

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.canvas,1,wx.EXPAND)
        sizer.Add(self.toolbar, 0 , wx.LEFT | wx.EXPAND)
        self.SetSizer(sizer)

class PlotNotebook(wx.Panel):
    def __init__(self, parent, id = -1):
        wx.Panel.__init__(self, parent, id=id)
        self.nb = wx.aui.AuiNotebook(self)
        sizer = wx.BoxSizer()
        sizer.Add(self.nb, 1, wx.EXPAND)
        self.SetSizer(sizer)

    def add(self,name="plot"):
        page = Plot(self.nb)
        self.nb.AddPage(page,name)
        return page.figure

def demo():
    app = wx.PySimpleApp()
    frame = wx.Frame(None,-1,'Plotter')
    plotter = PlotNotebook(frame)
    axes1 = plotter.add('figure 1').gca()
    axes1.plot([1,2,3],[2,1,4])
    axes2 = plotter.add('figure 2').gca()
    axes2.plot([1,2,3,4,5],[2,1,4,2,3])
    #axes1.figure.canvas.draw()
    #axes2.figure.canvas.draw()
    frame.Show()
    app.MainLoop()

if __name__ == "__main__": demo()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.13 user_interfaces example code: embedding_webagg.py

[source code]

This example demonstrates how to embed matplotlib WebAgg interactive
import io
try:
    import tornado
except ImportError:
    raise RuntimeError("This example requires tornado.")
import tornado.web
import tornado.httpserver
import tornado.ioloop
import tornado.websocket

from matplotlib.backends.backend_webagg_core import (FigureManagerWebAgg, new_figure_manager_given_figure)
from matplotlib.figure import Figure

import numpy as np
import json

def create_figure():
    """
    Creates a simple example figure.
    """
    fig = Figure()
a = fig.add_subplot(111)
t = np.arange(0.0, 3.0, 0.01)
s = np.sin(2 * np.pi * t)
a.plot(t, s)
return fig

# The following is the content of the web page. You would normally
# generate this using some sort of template facility in your web
# framework, but here we just use Python string formatting.
html_content = """
<html>
<head>
    <!-- TODO: There should be a way to include all of the required javascript
    and CSS so matplotlib can add to the set in the future if it
    needs to. -->
    <link rel="stylesheet" href="_static/css/page.css" type="text/css">
    <link rel="stylesheet" href="_static/css/boilerplate.css" type="text/css" />
    <link rel="stylesheet" href="_static/css/fbm.css" type="text/css" />
"""
class MyApplication(tornado.web.Application):
    class MainPage(tornado.web.RequestHandler):
        def get(self):

Serves the main HTML page.
manager = self.application.manager
ws_uri = "ws://{}\{req.host}\".format(req=self.request)
content = html_content % {
    "ws_uri": ws_uri, "fig_id": manager.num}
self.write(content)

class MplJs(tornado.web.RequestHandler):
    ""
    Serves the generated matplotlib javascript file. The content
    is dynamically generated based on which toolbar functions the
    user has defined. Call 'FigureManagerWebAgg' to get its
    content.
    ""
    def get(self):
        self.set_header('Content-Type', 'application/javascript')
        js_content = FigureManagerWebAgg.get_javascript()

        self.write(js_content)

class Download(tornado.web.RequestHandler):
    ""
    Handles downloading of the figure in various file formats.
    ""
    def get(self, fmt):
        manager = self.application.manager

        mimetypes = {
            'ps': 'application/postscript',
            'eps': 'application/postscript',
            'pdf': 'application/pdf',
            'svg': 'image/svg+xml',
            'png': 'image/png',
            'jpeg': 'image/jpeg',
            'tif': 'image/tiff',
            'emf': 'application/emf'
        }

        self.set_header('Content-Type', mimetypes.get(fmt, 'binary'))

        buff = io.BytesIO()
        manager.canvas.print_figure(buff, format=fmt)
        self.write(buff.getvalue())

class WebSocket(tornado.websocket.WebSocketHandler):
    ""
    A websocket for interactive communication between the plot in
    the browser and the server.

    In addition to the methods required by tornado, it is required to
    have two callback methods:

    - `send_json(json_content)` is called by matplotlib when
      it needs to send json to the browser. `json_content` is
Matplotlib, Release 1.4.3

a JSON tree (Python dictionary), and it is the responsibility of this implementation to encode it as a string to send over the socket.

```
send_binary(blob)
```

is called to send binary image data to the browser.

```python
supports_binary = True

def open(self):
    # Register the websocket with the FigureManager.
    manager = self.application.manager
    manager.add_web_socket(self)
    if hasattr(self, 'set_nodelay'):
        self.set_nodelay(True)

def on_close(self):
    # When the socket is closed, deregister the websocket with
    # the FigureManager.
    manager = self.application.manager
    manager.remove_web_socket(self)

def on_message(self, message):
    # The 'supports_binary' message is relevant to the
    # websocket itself. The other messages get passed along
    # to matplotlib as-is.

    message = json.loads(message)
    if message['type'] == 'supports_binary':
        self.supports_binary = message['value']
    else:
        manager = self.application.manager
        manager.handle_json(message)

def send_json(self, content):
    self.write_message(json.dumps(content))

def send_binary(self, blob):
    if self.supports_binary:
        self.write_message(blob, binary=True)
    else:
        data_uri = "data:image/png;base64,{}".format(blob.encode('base64').replace('\n', ' '))
        self.write_message(data_uri)

def __init__(self, figure):
    self.figure = figure
    self.manager = new_figure_manager_given_figure(id(figure), figure)

    super(MyApplication, self).__init__(
        # Static files for the CSS and JS

95.13. user_interfaces example code: embedding_webagg.py 2475
if __name__ == "__main__":
    figure = create_figure()
    application = MyApplication(figure)

    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(8080)

    print("http://127.0.0.1:8080/")
    print("Press Ctrl+C to quit")
    tornado.ioloop.IOLoop.instance().start()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.14 user Interfaces example code: fourier_demo_wx.py

[source code]

import numpy as np
import wx

import matplotlib
matplotlib.interactive(False)
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg
from matplotlib.figure import Figure
from matplotlib.pyplot import gcf, setp

class Knob:
    ""
    Knob - simple class with a "setKnob" method.
    A Knob instance is attached to a Param instance, e.g., param.attach(knob)""
Base class is for documentation purposes.

```python
def setKnob(self, value):
    pass
```

class Param:

```python
The idea of the "Param" class is that some parameter in the GUI may have several knobs that both control it and reflect the parameter's state, e.g. a slider, text, and dragging can all change the value of the frequency in the waveform of this example.

The class allows a cleaner way to update/"feedback" to the other knobs when one is being changed. Also, this class handles min/max constraints for all the knobs.

Idea - knob list - in "set" method, knob object is passed as well
- the other knobs in the knob list have a "set" method which gets called for the others.

```python
def __init__(self, initialValue=None, minimum=0., maximum=1.):
    self.minimum = minimum
    self.maximum = maximum
    if initialValue != self.constrain(initialValue):
        raise ValueError('illegal initial value')
    self.value = initialValue
    self.knobs = []

def attach(self, knob):
    self.knobs += [knob]

def set(self, value, knob=None):
    self.value = value
    self.value = self.constrain(value)
    for feedbackKnob in self.knobs:
        if feedbackKnob != knob:
            feedbackKnob.setKnob(self.value)
    return self.value

def constrain(self, value):
    if value <= self.minimum:
        value = self.minimum
    if value >= self.maximum:
        value = self.maximum
    return value
```

class SliderGroup(Param):
```python
def __init__(self, parent, label, param):
    self.sliderLabel = wx.StaticText(parent, label=label)
    self.sliderText = wx.TextCtrl(parent, -1, style=wx.TE_PROCESS_ENTER)
    self.slider = wx.Slider(parent, -1)
    self.slider.SetMax(param.maximum*1000)
    self.setKnob(param.value)
sizer = wx.BoxSizer(wx.HORIZONTAL)
sizer.Add(self.sliderLabel, 0, wx.EXPAND | wx.ALIGN_CENTER | wx.ALL, border=2)
sizer.Add(self.sliderText, 0, wx.EXPAND | wx.ALIGN_CENTER | wx.ALL, border=2)
sizer.Add(self.slider, 1, wx.EXPAND)
self.sizer = sizer

self.slider.Bind(wx.EVT_SLIDER, self.sliderHandler)
self.sliderText.Bind(wx.EVT_TEXT_ENTER, self.sliderTextHandler)

self.param = param
self.param.attach(self)

def sliderHandler(self, evt):
    value = evt.GetInt() / 1000.
    self.param.set(value)

def sliderTextHandler(self, evt):
    value = float(self.sliderText.GetValue())
    self.param.set(value)

def setKnob(self, value):
    self.sliderText.SetValue('%g' % value)
    self.slider.SetValue(value*1000)

class FourierDemoFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        self.fourierDemoWindow = FourierDemoWindow(self)
        self.frequencySliderGroup = SliderGroup(self, label='Frequency f0:', param=self.fourierDemoWindow.f0)
        self.amplitudeSliderGroup = SliderGroup(self, label='Amplitude a:', param=self.fourierDemoWindow.A)

        sizer = wx.BoxSizer(wx.VERTICAL)
sizer.Add(self.fourierDemoWindow, 1, wx.EXPAND)
sizer.Add(self.frequencySliderGroup.sizer, 0, wx.EXPAND | wx.ALIGN_CENTER | wx.ALL, border=5)
sizer.Add(self.amplitudeSliderGroup.sizer, 0, wx.EXPAND | wx.ALIGN_CENTER | wx.ALL, border=5)
self.SetSizer(sizer)

class FourierDemoWindow(wx.Window, Knob):
    def __init__(self, *args, **kwargs):
        wx.Window.__init__(self, *args, **kwargs)

        self.lines = []
        self.figure = Figure()
        self.canvas = FigureCanvasWxAgg(self, -1, self.figure)
        self.canvas.callbacks.connect('button_press_event', self.mouseDown)
        self.canvas.callbacks.connect('motion_notify_event', self.mouseMotion)
        self.canvas.callbacks.connect('button_release_event', self.mouseUp)
self.state = ''
self.mouseInfo = (None, None, None, None)
self.f0 = Param(2., minimum=0., maximum=6.)
self.A = Param(1., minimum=0.01, maximum=2.)
self.draw()

# Not sure I like having two params attached to the same Knob,
# but that is what we have here... it works but feels kludgy -
# although maybe it's not too bad since the knob changes both params
# at the same time (both f0 and A are affected during a drag)
self.f0.attach(self)
self.A.attach(self)
self.Bind(wx.EVT_SIZE, self.sizeHandler)

def sizeHandler(self, *args, **kwargs):
    self.canvas.SetSize(self.GetSize())

def mouseDown(self, evt):
    if self.lines[0] in self.figure.hitlist(evt):
        self.state = 'frequency'
    elif self.lines[1] in self.figure.hitlist(evt):
        self.state = 'time'
    else:
        self.state = ''
    self.mouseInfo = (evt.xdata, evt.ydata, max(self.f0.value, .1), self.A.value)

def mouseMotion(self, evt):
    if self.state == '':
        return
    x, y = evt.xdata, evt.ydata
    if x is None:
        # outside the axes
        return
    x0, y0, f0Init, AInit = self.mouseInfo
    self.A.set(AInit+(AInit*(y-y0)/y0), self)
    if self.state == 'frequency':
        self.f0.set(f0Init+(f0Init*(x-x0)/x0))
    elif self.state == 'time':
        if (x-x0)/x0 != -1.:
            self.f0.set(1./(1./f0Init+(1./f0Init*(x-x0)/x0)))

def mouseUp(self, evt):
    self.state = ''

def draw(self):
    if not hasattr(self, 'subplot1'):
        self.subplot1 = self.figure.add_subplot(211)
        self.subplot2 = self.figure.add_subplot(212)
    x1, y1, x2, y2 = self.compute(self.f0.value, self.A.value)
    color = (1., 0., 0.)
    self.lines += self.subplot1.plot(x1, y1, color=color, linewidth=2)
    self.lines += self.subplot2.plot(x2, y2, color=color, linewidth=2)
    # Set some plot attributes
    self.subplot1.set_title("Click and drag waveforms to change frequency and amplitude", fontsize=12)
self.subplot1.set_ylabel("Frequency Domain Waveform X(f)", fontsize = 8)
self.subplot1.set_xlabel("frequency f", fontsize = 8)
self.subplot2.set_ylabel("Time Domain Waveform x(t)", fontsize = 8)
self.subplot1.set_xlim([-6, 6])
self.subplot1.set_ylim([0, 1])
self.subplot2.set_xlim([-2, 2])
self.subplot2.set_ylim([-2, 2])
self.subplot1.text(0.05, .95, r'$X(f) = \mathcal{F}\{x(t)\}$', 
verticalalignment='top', transform = self.subplot1.transAxes)
self.subplot2.text(0.05, .95, r'$x(t) = a \cdot \cos(2\pi f_0 t) e^{-\pi t^2}$', 
verticalalignment='top', transform = self.subplot2.transAxes)

def compute(self, f0, A):
f = np.arange(-6., 6., 0.02)
t = np.arange(-2., 2., 0.01)
x = A*np.cos(2*np.pi*f0*t)*np.exp(-np.pi*t**2)
X = A/2*(np.exp(-np.pi*(f-f0)**2) + np.exp(-np.pi*(f+f0)**2))
return f, X, t, x

def repaint(self):
self.canvas.draw()

def setKnob(self, value):
    # Note, we ignore value arg here and just go by state of the params
    x1, y1, x2, y2 = self.compute(self.f0.value, self.A.value)
    setp(self.lines[0], xdata=x1, ydata=y1)
    setp(self.lines[1], xdata=x2, ydata=y2)
    self.repaint()

class App(wx.App):
    def OnInit(self):
        self.frame1 = FourierDemoFrame(parent=None, title="Fourier Demo", size=(640, 480))
        self.frame1.Show()
        return True

app = App()
app.MainLoop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.15 user_interfaces example code: gtk_spreadsheet.py

[source code]

#!/usr/bin/env python

"""
Example of embedding matplotlib in an application and interacting with 
a treeview to store data. Double click on an entry to update plot data
"""
```python
import pygtk
pygtk.require('2.0')
import gtk
from gtk import gdk

import matplotlib
matplotlib.use('GTKAgg')  # or 'GTK'
from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas

from numpy.random import random
from matplotlib.figure import Figure

class DataManager(gtk.Window):
    numRows, numCols = 20, 10
    data = random((numRows, numCols))

def __init__(self):
    gtk.Window.__init__(self)
    self.set_default_size(600, 600)
    self.connect('destroy', lambda win: gtk.main_quit())
    self.set_title('GtkListStore demo')
    self.set_border_width(8)

    vbox = gtk.VBox(False, 8)
    self.add(vbox)
    label = gtk.Label('Double click a row to plot the data')
    vbox.pack_start(label, False, False)

    sw = gtk.ScrolledWindow()
    sw.set_shadow_type(gtk.SHADOW_ETCHED_IN)
    sw.set_policy(gtk.POLICY_NEVER, gtk.POLICY_AUTOMATIC)
    vbox.pack_start(sw, True, True)

    model = self.create_model()
    self.treeview = gtk.TreeView(model)
    self.treeview.set_rules_hint(True)

    # matplotlib stuff
    fig = Figure(figsize=(6,4))
    self.canvas = FigureCanvas(fig)  # a gtk.DrawingArea
    vbox.pack_start(self.canvas, True, True)
    ax = fig.add_subplot(111)
    self.line, = ax.plot(self.data[0, :], 'go')  # plot the first row

```

95.15. user_interfaces example code: gtk_spreadsheet.py 2481
self.treeview.connect('row-activated', self.plot_row)
sw.add(self.treeview)

self.add_columns()

self.add_events(gdk.BUTTON_PRESS_MASK |
                gdk.KEY_PRESS_MASK|
                gdk.KEY_RELEASE_MASK)

def plot_row(self, treeview, path, view_column):
    ind, = path  # get the index into data
    points = self.data[ind, :]
    self.line.set_ydata(points)
    self.canvas.draw()

def add_columns(self):
    for i in range(self.numCols):
        column = gtk.TreeViewColumn('%d' % i, gtk.CellRendererText(), text=i)
        self.treeview.append_column(column)

def create_model(self):
    types = [float]*self.numCols
    store = gtk.ListStore(*types)

    for row in self.data:
        store.append(row)
    return store

manager = DataManager()
manager.show_all()
gtk.main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.16 user_interfaces example code: histogram_demo_canvasagg.py
[source code]

#!/usr/bin/env python

"""
This is an example that shows you how to work directly with the agg figure canvas to create a figure using the pythonic API.

In this example, the contents of the agg canvas are extracted to a string, which can in turn be passed off to PIL or put in a numeric array
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
from matplotlib.mlab import normpdf
from numpy.random import randn
import numpy

fig = Figure(figsize=(5,4), dpi=100)
ax = fig.add_subplot(111)
canvas = FigureCanvasAgg(fig)

mu, sigma = 100, 15
x = mu + sigma*randn(10000)

# the histogram of the data
n, bins, patches = ax.hist(x, 50, normed=1)

# add a 'best fit' line
y = normpdf(bins, mu, sigma)
line, = ax.plot(bins, y, 'r--')
line.set_linewidth(1)

ax.set_xlabel('Smarts')
ax.set_ylabel('Probability')
ax.set_title(r'$\mathrm{Histogram of IQ: }\mu=100, \sigma=15$')
ax.set_xlim( (40, 160))
ax.set_ylim( (0, 0.03))
canvas.draw()

s = canvas.tostring_rgb()  # save this and convert to bitmap as needed

# get the figure dimensions for creating bitmaps or numpy arrays, etc.
l,b,w,h = fig.bbox.bounds
w, h = int(w), int(h)

if 0:
    # convert to a numpy array
    X = numpy.fromstring(s, numpy.uint8)
    X.shape = h, w, 3

if 0:
    # pass off to PIL
    from PIL import Image
    im = Image.fromstring( "RGB", (w,h), s)
    im.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Matplotlib, Release 1.4.3

95.17 user_interfaces example code: interactive.py

[source code]

#!/usr/bin/env python

"""Multithreaded interactive interpreter with GTK and Matplotlib support.

WARNING:
As of 2010/06/25, this is not working, at least on Linux.
I have disabled it as a runnable script. - EF

Usage:

  pyint-gtk.py -> starts shell with gtk thread running separately

  pyint-gtk.py -pylab [filename] -> initializes matplotlib, optionally running
  the named file. The shell starts after the file is executed.

Threading code taken from:
http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/65109, by Brian
McErlean and John Finlay.

Matplotlib support taken from interactive.py in the matplotlib distribution.

Also borrows liberally from code.py in the Python standard library."""

from __future__ import print_function
__author__ = "Fernando Perez <Fernando.Perez@colorado.edu>"

import sys
import code
import threading
import gobject
import gtk

try:
    import readline
except ImportError:
    has_readline = False
else:
    has_readline = True

class MTConsole(code.InteractiveConsole):
    """Simple multi-threaded shell""

    def __init__(self, on_kill=None, *args, **kw):
        code.InteractiveConsole.__init__(self, *args, **kw)
        self.code_to_run = None
        self.ready = threading.Condition()
self._kill = False
if on_kill is None:
    on_kill = []
# Check that all things to kill are callable:
for _ in on_kill:
    if not callable(_):
        raise TypeError, 'on_kill must be a list of callables'
self.on_kill = on_kill
# Set up tab-completer
if has_readline:
    import rlcompleter
    try:
        # this form only works with python 2.3
        self.completer = rlcompleter.Completer(self.locals)
    except:
        # simpler for py2.2
        self.completer = rlcompleter.Completer()
    readline.set_completer(self.completer.complete)
    # Use tab for completions
    readline.parse_and_bind('tab: complete')
    # This forces readline to automatically print the above list when tab
    # completion is set to 'complete'.
    readline.parse_and_bind('set show-all-if-ambiguous on')
    # Bindings for incremental searches in the history. These searches
    # use the string typed so far on the command line and search
    # anything in the previous input history containing them.
    readline.parse_and_bind('\C-r': reverse-search-history')
    readline.parse_and_bind('\C-s': forward-search-history')

def runsoure(self, source, filename="<input>", symbol="single"):
    """Compile and run some source in the interpreter."

    Arguments are as for compile_command().

    One several things can happen:

    1) The input is incorrect; compile_command() raised an
       exception (SyntaxError or OverflowError). A syntax traceback
       will be printed by calling the showsyntaxerror() method.

    2) The input is incomplete, and more input is required;
       compile_command() returned None. Nothing happens.

    3) The input is complete; compile_command() returned a code
       object. The code is executed by calling self.runcode() (which
       also handles run-time exceptions, except for SystemExit).

    The return value is True in case 2, False in the other cases (unless
    an exception is raised). The return value can be used to
decide whether to use sys.ps1 or sys.ps2 to prompt the next
    line.
    """
    try:
        code = self.compile(source, filename, symbol)
except (OverflowError, SyntaxError, ValueError):
    # Case 1
    self.showsyntaxerror(filename)
    return False

if code is None:
    # Case 2
    return True

# Case 3
# Store code in self, so the execution thread can handle it
self.ready.acquire()
self.code_to_run = code
self.ready.wait()  # Wait until processed in timeout interval
self.ready.release()

return False

def runcode(self):
    """Execute a code object.

    When an exception occurs, self.showtraceback() is called to display a
    traceback.""

    self.ready.acquire()
    if self._kill:
        print('Closing threads...')
        sys.stdout.flush()
        for tokill in self.on_kill:
            tokill()
        print('Done. ')

    if self.code_to_run is not None:
        self.ready.notify()
        code.InteractiveConsole.runcode(self, self.code_to_run)

    self.code_to_run = None
    self.ready.release()
    return True

def kill(self):
    """Kill the thread, returning when it has been shut down.""
    self.ready.acquire()
    self._kill = True
    self.ready.release()

class GTKInterpreter(threading.Thread):
    """Run gtk.main in the main thread and a python interpreter in a
    separate thread.

    Python commands can be passed to the thread where they will be executed.
    This is implemented by periodically checking for passed code using a
    GTK timeout callback."""
TIMEOUT = 100 # Millisecond interval between timeouts.

def __init__(self, banner=None):
    threading.Thread.__init__(self)
    self.banner = banner
    self.shell = MTConsole(on_kill=[gtk.main_quit])

def run(self):
    self.pre_interact()
    self.shell.interact(self.banner)
    self.shell.kill()

def mainloop(self):
    self.start()
    gobject.timeout_add(self.TIMEOUT, self.shell.runcode)
    try:
        if gtk.gtk_version[0] >= 2:
            gtk.gdk.threads_init()
    except AttributeError:
        pass
    gtk.main()
    self.join()

def pre_interact(self):
    """This method should be overridden by subclasses.
    It gets called right before interact(), but after the thread starts.
    Typically used to push initialization code into the interpreter""
    pass

class MatplotlibInterpreter(GTKInterpreter):
    """Threaded interpreter with matplotlib support.

    Note that this explicitly sets GTKAgg as the backend, since it has
    specific GTK hooks in it.""

def __init__(self, banner=None):
    banner = """\nWelcome to matplotlib, a MATLAB-like python environment.\nhelp(matlab) -> help on matlab compatible commands from matplotlib.\nhelp(plotting) -> help on plotting commands.\n""
    GTKInterpreter.__init__(self, banner)

def pre_interact(self):
    """Initialize matplotlib before user interaction begins""
    push = self.shell.push
    # Code to execute in user's namespace
    lines = ['import matplotlib',
             'matplotlib.use('GTKAgg')",
             'matplotlib.interactive(1)",
             'import matplotlib.pylab as pylab",
             'import']
"from matplotlib.pylab import *
"

map(push,lines)

# Execute file if given.
if len(sys.argv)>1:
    import matplotlib
    matplotlib.interactive(0) # turn off interaction
    fname = sys.argv[1]
    try:
        inFile = file(fname, 'r')
    except IOError:
        print('*** ERROR *** Could not read file <%(s)>' % fname)
    else:
        print('*** Executing file <%(s)>:' % fname)
        for line in inFile:
            if line.lstrip().find('show()')==0: continue
            print('>>', line)
            push(line)
    inFile.close()
    matplotlib.interactive(1) # turn on interaction

if __name__ == '__main__':
    print("This demo is not presently functional, so running")
    print("it as a script has been disabled.")
    sys.exit()
    # Quick sys.argv hack to extract the option and leave filenames in sys.argv.
    # For real option handling, use optparse or getopt.
    if len(sys.argv) > 1 and sys.argv[1]=='-pylab':
        sys.argv = [sys.argv[0]]+sys.argv[2:]
        MatplotlibInterpreter().mainloop()
    else:
        GTKInterpreter().mainloop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.18 user Interfaces example code: interactive2.py

[source code]

#!/usr/bin/env python

from __future__ import print_function

# GTK Interactive Console
# (C) 2003, Jon Anderson
# See www.python.org/2.2/license.html for
# license details.
# import gtk
import gtk.gdk
import code
import os, sys
import pango
import __builtin__
import __main__

banner = """"GTK Interactive Python Console
Thanks to Jon Anderson
%s
"""" % sys.version

banner += """
Welcome to matplotlib.

  help(matplotlib) -- some general information about matplotlib
  help(plotting) -- shows a list of plot specific commands

"""

class Completer:
  """
  Taken from rlcompleter, with readline references stripped, and a local dictionary to use.
  """
  def __init__(self, locals):
    self.locals = locals

  def complete(self, text, state):
    """Return the next possible completion for 'text'.
    This is called successively with state == 0, 1, 2, ... until it
    returns None. The completion should begin with 'text'.
    """
    if state == 0:
      if "." in text:
        self.matches = self.attr_matches(text)
      else:
        self.matches = self.global_matches(text)
      try:
        return self.matches[state]
      except IndexError:
        return None

  def global_matches(self, text):
    """Compute matches when text is a simple name.

    Return a list of all keywords, built-in functions and names
    currently defines in __main__ that match.
    """

    import keyword
    matches = []
n = len(text)
for list in [keyword.kwlist, __builtin__.__dict__.keys(), __main__.__dict__.keys(), self.locals.keys()]
    for word in list:
        if word[:n] == text and word != "__builtins__":
            matches.append(word)
return matches

def attr_matches(self, text):
    
    
    Assuming the text is of the form NAME.NAME....[NAME], and is evaluable in the globals of __main__, it will be evaluated and its attributes (as revealed by dir()) are used as possible completions. (For class instances, class members are also considered.)

    WARNING: this can still invoke arbitrary C code, if an object with a __getattr__ hook is evaluated.

    
    import re
    m = re.match(r"(\w+(\.\w+)*).\w*", text)
    if not m:
        return
    expr, attr = m.group(1, 3)
    object = eval(expr, __main__.__dict__, self.locals)
    words = dir(object)
    if hasattr(object, '__class__'):
        words.append('__class__')
        words = words + get_class_members(object.__class__)
    matches = []
    n = len(attr)
    for word in words:
        if word[:n] == attr and word != "__builtins__":
            matches.append("%s.%s" % (expr, word))
    return matches

def get_class_members(klass):
    ret = dir(klass)
    if hasattr(klass, '__bases__'):
        for base in klass.__bases__:
            ret = ret + get_class_members(base)
    return ret

class OutputStream:
    
    A Multiplexing output stream.
    It can replace another stream, and tee output to the original stream and too a GTK textview.
    
    def __init__(self, view, old_out, style):
self.view = view
self.buffer = view.get_buffer()
self.mark = self.buffer.create_mark("End", self.buffer.get_end_iter(), False)
self.out = old_out
self.style = style
self.tee = 1

def write(self, text):
    if self.tee:
        self.out.write(text)
    end = self.buffer.get_end_iter()
    if not self.view == None:
        self.view.scroll_to_mark(self.mark, 0, True, 1, 1)
    self.buffer.insert_with_tags(end, text, self.style)

class GTKInterpreterConsole(gtk.ScrolledWindow):
    ""
    An InteractiveConsole for GTK. It's an actual widget, so it can be dropped in just about anywhere.
    ""
    def __init__(self):
        gtk.ScrolledWindow.__init__(self)
        self.set_policy (gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)
        self.text = gtk.TextView()
        self.text.set_wrap_mode(True)
        self.interpreter = code.InteractiveInterpreter()
        self.completer = Completer(self.interpreter.locals)
        self.buffer = []
        self.history = []
        self.banner = banner
        self.ps1 = ">>> ">
        self.ps2 = "... ">
        self.text.add_events(gtk.gdk.KEY_PRESS_MASK)
        self.text.connect("key_press_event", self.key_pressed)
        self.current_history = -1
        self.mark = self.text.get_buffer().create_mark("End", self.text.get_buffer().get_end_iter(), False)

    #setup colors
    self.style_banner = gtk.TextTag("banner")
    self.style_banner.set_property("foreground", "saddle brown")
    self.style_ps1 = gtk.TextTag("ps1")
    self.style_ps1.set_property("foreground", "DarkOrchid4")
    self.style_ps1.set_property("editable", False)
self.style_ps1.set_property("font", "courier")

self.style_ps2 = gtk.TextTag("ps2")
self.style_ps2.set_property( "foreground", "DarkOliveGreen" )
self.style_ps2.set_property( "editable", False )
self.style_ps2.set_property("font", "courier")

self.style_out = gtk.TextTag("stdout")
self.style_out.set_property( "foreground", "midnight blue" )
self.style_err = gtk.TextTag("stderr")
self.style_err.set_property( "style", pango.STYLE_ITALIC )
self.style_err.set_property( "foreground", "red" )

self.text.get_buffer().get_tag_table().add(self.style_banner)
self.text.get_buffer().get_tag_table().add(self.style_ps1)
self.text.get_buffer().get_tag_table().add(self.style_ps2)
self.text.get_buffer().get_tag_table().add(self.style_out)
self.text.get_buffer().get_tag_table().add(self.style_err)

self.stdout = OutputStream(self.text,sys.stdout,self.style_out)
self.stderr = OutputStream(self.text,sys.stderr,self.style_err)

sys.stderr = self.stderr
sys.stdout = self.stdout

self.current_prompt = None

self.write_line(self.banner, self.style_banner)
sel.prompt_ps1()

self.add(self.text)
self.text.show()

def reset_history(self):
    self.history = []

def reset_buffer(self):
    self.buffer = []

def prompt_ps1(self):
    self.current_prompt = self.prompt_ps1
    self.write_line(self.ps1,self.style_ps1)

def prompt_ps2(self):
    self.current_prompt = self.prompt_ps2
    self.write_line(self.ps2,self.style_ps2)

def write_line(self,text,style=None):
    start,end = self.text.get_buffer().get_bounds()
    if style==None:
        self.text.get_buffer().insert(end,text)
    else:
self.text.get_buffer().insert_with_tags(end, text, style)

self.text.scroll_to_mark(self.mark, 0, True, 1, 1)

def push(self, line):

    self.buffer.append(line)
    if len(line) > 0:
        self.history.append(line)

    source = "\n".join(self.buffer)

    more = self.interpreter.runsource(source, "<<console>>")

    if not more:
        self.reset_buffer()
        return
    more

def key_pressed(self, widget, event):

    if event.keyval == gtk.gdk.keyval_from_name('Return'):
        return self.execute_line()

    if event.keyval == gtk.gdk.keyval_from_name('Up'):
        self.current_history = self.current_history - 1
        if self.current_history < -len(self.history):
            self.current_history = -len(self.history)
        return self.show_history()

    elif event.keyval == gtk.gdk.keyval_from_name('Down'):
        self.current_history = self.current_history + 1
        if self.current_history > 0:
            self.current_history = 0
        return self.show_history()

    elif event.keyval == gtk.gdk.keyval_from_name('Home'):
        l = self.text.get_buffer().get_line_count() - 1
        start = self.text.get_buffer().get_iter_at_line_offset(l, 1)
        self.text.get_buffer().place_cursor(start)
        return True

    elif event.keyval == gtk.gdk.keyval_from_name('space')
        and event.state & gtk.gdk.CONTROL_MASK:
        return self.complete_line()

    return False

def show_history(self):

    if self.current_history == 0:
        return True
    else:
        self.replace_line(self.history[self.current_history])
        return True

def current_line(self):

    start, end = self.current_line_bounds()
    return self.text.get_buffer().get_text(start, end, True)
```python
def current_line_bounds(self):
    txt_buffer = self.text.get_buffer()
    l = txt_buffer.get_line_count() - 1
    start = txt_buffer.get_iter_at_line(l)
    if start.get_chars_in_line() >= 4:
        start.forward_chars(4)
    end = txt_buffer.get_end_iter()
    return start, end

def replace_line(self, txt):
    start, end = self.current_line_bounds()
    self.text.get_buffer().delete(start, end)
    self.write_line(txt)

def execute_line(self, line=None):
    if line is None:
        line = self.current_line()
        self.write_line("\n")
    else:
        self.write_line(line + "\n")

    more = self.push(line)

    self.text.get_buffer().place_cursor(self.text.get_buffer().get_end_iter())

    if more:
        self.prompt_ps2()
    else:
        self.prompt_ps1()

    self.current_history = 0
    self.window.raise_()

    return True

def complete_line(self):
    line = self.current_line()
    tokens = line.split()
    token = tokens[-1]

    completions = []
    p = self.completer.complete(token, len(completions))
    while p != None:
        completions.append(p)
        p = self.completer.complete(token, len(completions))

    if len(completions) != 1:
        self.write_line("\n")
        self.write_line("\n".join(completions), self.style_ps1)
```

Chapter 95. user_interfaces Examples
```python
self.write_line("\n")
self.current_prompt()
self.write_line(line)
else:
i = line.rfind(token)
line = line[0:i] + completions[0]
self.replace_line(line)

return True

def main():
w = gtk.Window()
console = GTKInterpreterConsole()
console.set_size_request(640,480)
w.add(console)

def destroy(arg=None):
gtk.main_quit()

def key_event(widget,event):
    if gtk.gdk.keyval_name( event.keyval) == 'd' and 
        event.state & gtk.gdk.CONTROL_MASK:
        destroy()
    return False
w.connect("destroy", destroy)

w.add_events( gtk.gdk.KEY_PRESS_MASK )
w.connect( 'key_press_event', key_event)
w.show_all()

console.execute_line('import matplotlib')
console.execute_line("matplotlib.use('GTKAgg')")
console.execute_line('matplotlib.interactive(1)')
console.execute_line('from pylab import *')

if len(sys.argv)>1:
    fname = sys.argv[1]
    if not os.path.exists(fname):
        print('%s does not exist' % fname)
    for line in file(fname):
        line = line.strip()

        console.execute_line(line)
gtk.main()

if __name__ == '__main__':
    main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
95.19 user_interfaces example code: lineprops_dialog_gtk.py

[source code]

```python
import matplotlib
matplotlib.use('GTKAgg')
from matplotlib.backends.backend_gtk import DialogLineprops

import numpy as np
import matplotlib.pyplot as plt

def f(t):
    s1 = np.cos(2*np.pi*t)
    e1 = np.exp(-t)
    return np.multiply(s1,e1)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

fig, ax = plt.subplots()
l1, = ax.plot(t1, f(t1), 'bo', label='line 1')
l2, = ax.plot(t2, f(t2), 'k--', label='line 2')

dlg = DialogLineprops([l1,l2])
dlg.show()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.20 user_interfaces example code: mathtext_wx.py

[source code]

```python
""
Demonstrates how to convert mathtext to a wx.Bitmap for display in various
controls on wxPython.
""

import matplotlib
matplotlib.use("WXAgg")
from numpy import arange, sin, pi, cos, log
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from matplotlib.figure import import Figure

import wx

IS_GTK = 'wxGTK' in wx.PlatformInfo
IS_WIN = 'wxMSW' in wx.PlatformInfo
```
IS_MAC = 'wxMac' in wx.PlatformInfo

# This is where the "magic" happens.
from matplotlib.mathtext import MathTextParser
mathtext_parser = MathTextParser("Bitmap")

def mathtext_to_wxbitmap(s):
    ftimage, depth = mathtext_parser.parse(s, 150)
    return wx.BitmapFromBufferRGBA(
        ftimage.get_width(), ftimage.get_height(),
        ftimage.as_rgba_str())

functions = [
    (r'$\sin(2 \pi x)$', lambda x: sin(2*pi*x)),
    (r'$\frac{4}{3}\pi x^3$', lambda x: (4.0 / 3.0) * pi * x**3),
    (r'$\cos(2 \pi x)$', lambda x: cos(2*pi*x)),
    (r'$\log(x)$', lambda x: log(x))
]

class CanvasFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, -1, title, size=(550, 350))
        self.SetBackgroundColour(wx.NamedColour("WHITE"))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)
        self.change_plot(0)

        self.canvas = FigureCanvas(self, -1, self.figure)
        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.add_buttonbar()
        self.add_toolbar()

        menuBar = wx.MenuBar()

        # File Menu
        menu = wx.Menu()
        menu.Append(wx.ID_EXIT, "E&xit	Alt-X", "Exit this simple sample")
        menuBar.Append(menu, "&File")

        if IS_GTK or IS_WIN:
            # Equation Menu
            menu = wx.Menu()
            for i, (mt, func) in enumerate(functions):
                bm = mathtext_to_wxbitmap(mt)
                item = wx.MenuItem(menu, 1000 + i, " ")
                item.SetBitmap(bm)
                menu.AppendItem(item)
                self.Bind(wx.EVT_MENU, selfOnChangePlot, item)
            menuBar.Append(menu, "&Functions")

95.20. user_interfaces example code: mathtext_wx.py 2497
self.SetMenuBar(menuBar)

self.SetSizer(self.sizer)
self.Fit()

def add_buttonbar(self):
    self.button_bar = wx.Panel(self)
    self.button_bar_sizer = wx.BoxSizer(wx.HORIZONTAL)
    self.sizer.Add(self.button_bar, 0, wx.LEFT | wx.TOP | wx.GROW)

    for i, (mt, func) in enumerate(functions):
        bm = mathtext_to_wxbitmap(mt)
        button = wx.BitmapButton(self.button_bar, 1000 + i, bm)
        self.button_bar_sizer.Add(button, 1, wx.GROW)
        self.Bind(wx.EVT_BUTTON, self.OnChangePlot, button)

    self.button_bar.SetSizer(self.button_bar_sizer)

def add_toolbar(self):
    """Copied verbatim from embedding_wx2.py"""
    self.toolbar = NavigationToolbar2Wx(self.canvas)
    self.toolbar.Realize()
    if IS_MAC:
        self.SetToolBar(self.toolbar)
    else:
        tw, th = self.toolbar.GetSizeTuple()
        fw, fh = self.canvas.GetSizeTuple()
        self.toolbar.SetSize(wx.Size(fw, th))
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
    self.toolbar.update()

def OnPaint(self, event):
    self.canvas.draw()

def OnChangePlot(self, event):
    self.change_plot(event.GetId() - 1000)

def change_plot(self, plot_number):
    t = arange(1.0, 3.0, 0.01)
    s = functions[plot_number][1](t)
    self.axes.clear()
    self.axes.plot(t, s)
    self.Refresh()

class MyApp(wx.App):
    def OnInit(self):
        frame = CanvasFrame(None, "wxPython mathtext demo app")
        self.SetTopWindow(frame)
        frame.Show(True)
        return True

app = MyApp()
app.MainLoop()
Importing necessary libraries and modules:

```python
from __future__ import print_function
import matplotlib
matplotlib.use('GTK')
```

Then, imports the specific modules and classes:

```python
from matplotlib.figure import Figure
from matplotlib.axes import Subplot
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as NavigationToolbar
from matplotlib.widgets import SpanSelector
from numpy import arange, sin, pi
import gtk
import gtk.glade
```

Next, define a function `simple_msg` for displaying messages:

```python
def simple_msg(msg, parent=None, title=None):
    dialog = gtk.MessageDialog(
        parent = None,
        type = gtk.MESSAGE_INFO,
        buttons = gtk.BUTTONS_OK,
        message_format = msg)
    if parent is not None:
        dialog.set_transient_for(parent)
    if title is not None:
        dialog.set_title(title)
    dialog.show()
    dialog.run()
    dialog.destroy()
    return None
```

Define a class `GladeHandlers` with a method `on_buttonClickMe_clicked`:

```python
class GladeHandlers:
    def on_buttonClickMe_clicked(event):
        simple_msg('Nothing to say, really', parent=widgets['windowMain'], title='Thanks!')
```

Define a class `WidgetsWrapper` with an initializer and a method `signal_autoconnect`:

```python
class WidgetsWrapper:
    def __init__(self):
        self.widgets = gtk.glade.XML('mpl_with_glade.glade')
        self.widgets.signal_autoconnect(GladeHandlers.__dict__)
```
self['windowMain'].connect('destroy', lambda x: gtk.main_quit())
self['windowMain'].move(10,10)
self.figure = Figure(figsize=(8,6), dpi=72)
self.axis = self.figure.add_subplot(111)

t = arange(0.0,3.0,0.01)
s = sin(2*pi*t)
self.axis.plot(t,s)
self.axis.set_xlabel('time (s)')
self.axis.set_ylabel('voltage')

self.canvas = FigureCanvas(self.figure)  # a gtk.DrawingArea
self.canvas.show()
self.canvas.set_size_request(600, 400)
self.canvas.set_events(gtk.gdk.BUTTON_PRESS_MASK |
                       gtk.gdk.KEY_PRESS_MASK |
                       gtk.gdk.KEY_RELEASE_MASK)
self.canvas.set_flags(gtk.HAS_FOCUS|gtk.CAN_FOCUS)
self.canvas.grab_focus()

def keypress(widget, event):
    print('key press')
def buttonpress(widget, event):
    print('button press')

self.canvas.connect('key_press_event', keypress)
self.canvas.connect('button_press_event', buttonpress)

def onselect(xmin, xmax):
    print(xmin, xmax)

span = SpanSelector(self.axis, onselect, 'horizontal', useblit=False,
                     rectprops=dict(alpha=0.5, facecolor='red'))

self['vboxMain'].pack_start(self.canvas, True, True)
self['vboxMain'].show()

# below is optional if you want the navigation toolbar
self.navToolbar = NavigationToolbar(self.canvas, self['windowMain'])
self.navToolbar.lastDir = '/var/tmp/
self['vboxMain'].pack_start(self.navToolbar)
self.navToolbar.show()

sep = gtk.HSeparator()
sep.show()
self['vboxMain'].pack_start(sep, True, True)

self['vboxMain'].reorder_child(self['buttonClickMe'],-1)
```python
def __getitem__(self, key):
    return self.widgets.get_widget(key)

widgets = WidgetsWrapper()
gtk.main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.22 user_interfaces example code: mpl_with_glade_316.py

[source code]

#!/usr/bin/env python3

from gi.repository import Gtk

from matplotlib.figure import Figure
from matplotlib.axes import Subplot
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas

class Window1Signals:
    def on_window1_destroy(self, widget):
        Gtk.main_quit()

def main():
    builder = Gtk.Builder()
    builder.add_objects_from_file("mpl_with_glade_316.glade", ("window1", "") )
    builder.connect_signals(Window1Signals())
    window = builder.get_object("window1")
    sw = builder.get_object("scrolledwindow1")

    #Start of Matplotlib specific code
    figure = Figure(figsize=(8,6), dpi=71)
    axis = figure.add_subplot(111)
    t = arange(0.0, 3.0, 0.01)
    s = sin(2*pi*t)
    axis.plot(t,s)

    axis.set_xlabel('time [s]')
    axis.set_ylabel('voltage [V]')

    canvas = FigureCanvas(figure)  # a Gtk.DrawingArea
    canvas.set_size_request(800,600)
    sw.add_with_viewport(canvas)

    #End of Matplotlib specific code
```
An example of how to use pylab to manage your figure windows, but modify the GUI by accessing the underlying gtk widgets

```python
from __future__ import print_function
import matplotlib
matplotlib.use('GTKAgg')
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
plt.plot([1,2,3], 'ro-', label='easy as 1 2 3')
plt.plot([1,4,9], 'gs--', label='easy as 1 2 3 squared')
plt.legend()

manager = plt.get_current_fig_manager()
# you can also access the window or vbox attributes this way
toolbar = manager.toolbar

# now let's add a button to the toolbar
import gtk
next = 8; #where to insert this in the mpl toolbar
button = gtk.Button('Click me')
button.show()

def clicked(button):
    print('hi mom')
button.connect('clicked', clicked)

toolitem = gtk.ToolItem()
toolitem.show()
toolitem.set_tooltip(toolbar.tooltips, 'Click me for fun and profit')
toolitem.add(button)
toolbar.insert(toolitem, next); next +=1
```

95.23 user_interfaces example code: pylab_with_gtk.py
Matplotlib, Release 1.4.3

# now let's add a widget to the vbox
label = gtk.Label()
label.set_markup('Drag mouse over axes for position')
label.show()
vbox = manager.vbox
vbox.pack_start(label, False, False)
vbox.reorder_child(manager.toolbar, -1)

def update(event):
    if event.xdata is None:
        label.set_markup('Drag mouse over axes for position')
    else:
        label.set_markup('<span color="#ef0000">x,y=(%f, %f)</span>' % (event.xdata, event.ydata))

plt.connect('motion_notify_event', update)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.24 user_interfaces example code: rec_edit_gtk_custom.py

[source code]

""
generate an editable gtk treeview widget for record arrays with custom
formatting of the cells and show how to limit string entries to a list
of strings
""

from __future__ import print_function
import gtk
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import mpl_toolkits.gtktools as gtktools

datafile = cbook.get_sample_data('demodata.csv', asfileobj=False)
r = mlab.csv2rec(datafile, converterd={'weekdays':str})

formatd = mlab.get_formatd(r)
formatd['date'] = mlab.FormatDate('%Y-%m-%d')
formatd['prices'] = mlab.FormatMillions(precision=1)
formatd['gain'] = mlab.FormatPercent(precision=2)

# use a drop down combo for weekdays
stringd = dict(weekdays=['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])
constant = ['clientid']  # block editing of this field

95.24. user_interfaces example code: rec_edit_gtk_custom.py 2503
liststore = gtktools.RecListStore(r, formatd=formatd, stringd=stringd)
treeview = gtktools.RecTreeView(liststore, constant=constant)

def mycallback(liststore, rownum, colname, oldval, newval):
    print("verify: old=%s, new=%s, rec=%s\n(\%s\)\n) % (oldval, newval, liststore.r[rownum][colname])

liststore.callbacks.connect('cell_changed', mycallback)

win = gtk.Window()
win.set_title('click to edit')
win.add(treeview)
win.show_all()
win.connect('delete-event', lambda *args: gtk.main_quit())
gtk.main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.25 user_interfaces example code: rec_edit_gtk_simple.py

[source code]

Load a CSV file into a record array and edit it in a gtk treeview

import gtk
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import mpl_toolkits.gtktools as gtktools

datafile = cbook.get_sample_data('demodata.csv', asfileobj=False)
r = mlab.csv2rec(datafile, converterd={'weekdays':'str'})

liststore, treeview, win = gtktools.edit_recarray(r)
win.set_title('click to edit')
win.connect('delete-event', lambda *args: gtk.main_quit())
gtk.main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.26 user_interfaces example code: svg_histogram.py

[source code]

#!/usr/bin/env python
#-*- encoding: utf-8 -*-
Demonstrate how to create an interactive histogram, in which bars are hidden or shown by clicking on legend markers.

The interactivity is encoded in ecmaScript (javascript) and inserted in the SVG code in a post-processing step. To render the image, open it in a web browser. SVG is supported in most web browsers used by Linux and OSX users. Windows IE9 supports SVG, but earlier versions do not.

Notes
-----
The matplotlib backend lets us assign ids to each object. This is the mechanism used here to relate matplotlib objects created in python and the corresponding SVG constructs that are parsed in the second step. While flexible, ids are cumbersome to use for large collection of objects. Two mechanisms could be used to simplify things:
* systematic grouping of objects into SVG <g> tags,
* assigning classes to each SVG object according to its origin.

For example, instead of modifying the properties of each individual bar, the bars from the 'hist' function could either be grouped in a PatchCollection, or be assigned a class="hist_##" attribute.

CSS could also be used more extensively to replace repetitive markup throughout the generated SVG.

__author__="david.huard@gmail.com"

import numpy as np
import matplotlib.pyplot as plt
import xml.etree.ElementTree as ET
from StringIO import StringIO
import json

plt.rcParams['svg.embed_char_paths'] = 'none'

# Apparently, this `register_namespace` method works only with
# python 2.7 and up and is necessary to avoid garbling the XML name
# space with ns0.
ET.register_namespace('',"http://www.w3.org/2000/svg")

# --- Create histogram, legend and title ---
plt.figure()
r = np.random.randn(100)
r1 = r + 1
labels = ['Rabbits', 'Frogs']
H = plt.hist([r,r1], label=labels)
containers = H[-1]
leg = plt.legend(frameon=False)
plt.title("""From a web browser, click on the legend marker to toggle the corresponding histogram."""")

# --- Add ids to the svg objects we'll modify

hist_patches = {}
for ic, c in enumerate(containers):
    hist_patches['hist_%d'%ic] = []
    for il, element in enumerate(c):
        element.set_gid('hist_%d_patch_%d'%(ic, il))
        hist_patches['hist_%d'%ic].append('hist_%d_patch_%d'%(ic, il))

# Set ids for the legend patches
for i, t in enumerate(leg.get_patches()):
    t.set_gid('leg_patch_%d'%i)

# Set ids for the text patches
for i, t in enumerate(leg.get_texts()):
    t.set_gid('leg_text_%d'%i)

# Save SVG in a fake file object.
f = StringIO()
plt.savefig(f, format="svg")

# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())

# --- Add interactivity ---

# Add attributes to the patch objects.
for i, t in enumerate(leg.get_patches()):
    el = xmlid['leg_patch_%d'%i]
    el.set('cursor', 'pointer')
    el.set('onclick', "toggle_hist(this)"

# Add attributes to the text objects.
for i, t in enumerate(leg.get_texts()):
    el = xmlid['leg_text_%d'%i]
    el.set('cursor', 'pointer')
    el.set('onclick', "toggle_hist(this)"

# Create script defining the function 'toggle_hist'.
# We create a global variable 'container' that stores the patches id
# belonging to each histogram. Then a function "toggle_element" sets the
# visibility attribute of all patches of each histogram and the opacity
# of the marker itself.

script = ""
<script type="text/ecmascript">

2506 Chapter 95. user_interfaces Examples
Matplotlib, Release 1.4.3

<![CDATA[
var container = %s
function toggle(oid, attribute, values) {
/* Toggle the style attribute of an object between two values.
Parameters
---------oid : str
Object identifier.
attribute : str
Name of syle attribute.
values : [on state, off state]
The two values that are switched between.
*/
var obj = document.getElementById(oid);
var a = obj.style[attribute];
a = (a == values[0] || a == "") ? values[1] : values[0];
obj.style[attribute] = a;
}
function toggle_hist(obj) {
var num = obj.id.slice(-1);
toggle('leg_patch_' + num, 'opacity', [1, 0.3]);
toggle('leg_text_' + num, 'opacity', [1, 0.5]);
var names = container['hist_'+num]
for (var i=0; i < names.length; i++) {
toggle(names[i], 'opacity', [1,0])
};
}
]]>
</script>
"""%json.dumps(hist_patches)

# Add a transition effect
css = tree.getchildren()[0][0]
css.text = css.text + "g {-webkit-transition:opacity 0.4s ease-out;-moz-transition:opacity 0.4s ease-out
# Insert the script and save to file.
tree.insert(0, ET.XML(script))
ET.ElementTree(tree).write("svg_histogram.svg")

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.26. user_interfaces example code: svg_histogram.py

2507


SVG tooltip example
===================

This example shows how to create a tooltip that will show up when hovering over a matplotlib patch.

Although it is possible to create the tooltip from CSS or javascript, here we create it in matplotlib and simply toggle its visibility on when hovering over the patch. This approach provides total control over the tooltip placement and appearance, at the expense of more code up front.

The alternative approach would be to put the tooltip content in `title` attributes of SVG objects. Then, using an existing js/CSS library, it would be relatively straightforward to create the tooltip in the browser. The content would be dictated by the `title` attribute, and the appearance by the CSS.

:author: David Huard

```python
import matplotlib.pyplot as plt
import xml.etree.ElementTree as ET
from StringIO import StringIO

ET.register_namespace('', 'http://www.w3.org/2000/svg')

fig, ax = plt.subplots()

# Create patches to which tooltips will be assigned.
circle = plt.Circle((0, 0), 5, fc='blue')
rect = plt.Rectangle((-5, 10), 10, 5, fc='green')

ax.add_patch(circle)
ax.add_patch(rect)

# Create the tooltips
circle_tip = ax.annotate('This is a blue circle.',
    xy=(0, 0),
    xytext=(30, -30),
    textcoords='offset points',
    color='w',
    ha='left',
    bbox=dict(boxstyle='round,pad=.5', fc=(.1,.1,.1,.92), ec=(1.,1.,1.), lw=1, zorder=1),
)
rect_tip = ax.annotate('This is a green rectangle.',
                      xy=(-5,10),
                      xytext=(30,40),
                      textcoords='offset points',
                      color='w',
                      ha='left',
                      bbox=dict(boxstyle='round,pad=.5',
                                fc=(.1,.1,.1,.92),
                                ec=(1.,1.,1.),
                                lw=1, zorder=1),
                      )

# Set id for the patches
for i, t in enumerate(ax.patches):
    t.set_gid('patch_%d'%i)

# Set id for the annotations
for i, t in enumerate(ax.texts):
    t.set_gid('tooltip_%d'%i)

# Save the figure in a fake file object
ax.set_xlim(-30, 30)
ax.set_ylim(-30, 30)
ax.set_aspect('equal')

f = StringIO()
plt.savefig(f, format="svg")

# --- Add interactivity ---
# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())
tree.set('onload', 'init(evt)')

# Hide the tooltips
for i, t in enumerate(ax.texts):
    el = xmlid['tooltip_%d'%i]
    el.set('visibility', 'hidden')

# Assign onmouseover and onmouseout callbacks to patches.
for i, t in enumerate(ax.patches):
    el = xmlid['patch_%d'%i]
    el.set('onmouseover', "ShowTooltip(this)"
    el.set('onmouseout', "HideTooltip(this)"

# This is the script defining the ShowTooltip and HideTooltip functions.
script = ""
<script type="text/ecmascript">
<![CDATA[

    function init(evt) {
        if ( window.svgDocument == null ) {
            svgDocument = evt.target.ownerDocument;
        }

    function show_tooltip(evt) {
        var tooltip = evt.target;
        tooltip.style.visibility = "visible";
    }

    function hide_tooltip(evt) {
        var tooltip = evt.target;
        tooltip.style.visibility = "hidden";
    }

    var els = svgDocument.querySelectorAll('tooltip');
    for (var i = 0; i < els.length; i++) {
        els[i].addEventListener('onmouseover', show_tooltip);
        els[i].addEventListener('onmouseout', hide_tooltip);
    }

    var patch_A = svgDocument.getElementById('patch_0');
    patch_A.addEventListener('onmouseover', show_tooltip);
    patch_A.addEventListener('onmouseout', hide_tooltip);

    var patch_B = svgDocument.getElementById('patch_1');
    patch_B.addEventListener('onmouseover', show_tooltip);
    patch_B.addEventListener('onmouseout', hide_tooltip);

    var patch_C = svgDocument.getElementById('patch_2');
    patch_C.addEventListener('onmouseover', show_tooltip);
    patch_C.addEventListener('onmouseout', hide_tooltip);

    var patch_D = svgDocument.getElementById('patch_3');
    patch_D.addEventListener('onmouseover', show_tooltip);
    patch_D.addEventListener('onmouseout', hide_tooltip);

    var patch_E = svgDocument.getElementById('patch_4');
    patch_E.addEventListener('onmouseover', show_tooltip);
    patch_E.addEventListener('onmouseout', hide_tooltip);

    var patch_F = svgDocument.getElementById('patch_5');
    patch_F.addEventListener('onmouseover', show_tooltip);
    patch_F.addEventListener('onmouseout', hide_tooltip);

    var patch_G = svgDocument.getElementById('patch_6');
    patch_G.addEventListener('onmouseover', show_tooltip);
    patch_G.addEventListener('onmouseout', hide_tooltip);

    var patch_H = svgDocument.getElementById('patch_7');
    patch_H.addEventListener('onmouseover', show_tooltip);
    patch_H.addEventListener('onmouseout', hide_tooltip);

    var patch_I = svgDocument.getElementById('patch_8');
    patch_I.addEventListener('onmouseover', show_tooltip);
    patch_I.addEventListener('onmouseout', hide_tooltip);

    var patch_J = svgDocument.getElementById('patch_9');
    patch_J.addEventListener('onmouseover', show_tooltip);
    patch_J.addEventListener('onmouseout', hide_tooltip);

    var patch_K = svgDocument.getElementById('patch_10');
    patch_K.addEventListener('onmouseover', show_tooltip);
    patch_K.addEventListener('onmouseout', hide_tooltip);

    var patch_L = svgDocument.getElementById('patch_11');
    patch_L.addEventListener('onmouseover', show_tooltip);
    patch_L.addEventListener('onmouseout', hide_tooltip);

    var patch_M = svgDocument.getElementById('patch_12');
    patch_M.addEventListener('onmouseover', show_tooltip);
    patch_M.addEventListener('onmouseout', hide_tooltip);

    var patch_N = svgDocument.getElementById('patch_13');
    patch_N.addEventListener('onmouseover', show_tooltip);
    patch_N.addEventListener('onmouseout', hide_tooltip);

    var patch_O = svgDocument.getElementById('patch_14');
    patch_O.addEventListener('onmouseover', show_tooltip);
    patch_O.addEventListener('onmouseout', hide_tooltip);

    var patch_P = svgDocument.getElementById('patch_15');
    patch_P.addEventListener('onmouseover', show_tooltip);
    patch_P.addEventListener('onmouseout', hide_tooltip);

    var patch_Q = svgDocument.getElementById('patch_16');
    patch_Q.addEventListener('onmouseover', show_tooltip);
    patch_Q.addEventListener('onmouseout', hide_tooltip);

    var patch_R = svgDocument.getElementById('patch_17');
    patch_R.addEventListener('onmouseover', show_tooltip);
    patch_R.addEventListener('onmouseout', hide_tooltip);

    var patch_S = svgDocument.getElementById('patch_18');
    patch_S.addEventListener('onmouseover', show_tooltip);
    patch_S.addEventListener('onmouseout', hide_tooltip);

    var patch_T = svgDocument.getElementById('patch_19');
    patch_T.addEventListener('onmouseover', show_tooltip);
    patch_T.addEventListener('onmouseout', hide_tooltip);

    var patch_U = svgDocument.getElementById('patch_20');
    patch_U.addEventListener('onmouseover', show_tooltip);
    patch_U.addEventListener('onmouseout', hide_tooltip);

    var patch_V = svgDocument.getElementById('patch_21');
    patch_V.addEventListener('onmouseover', show_tooltip);
    patch_V.addEventListener('onmouseout', hide_tooltip);

    var patch_W = svgDocument.getElementById('patch_22');
    patch_W.addEventListener('onmouseover', show_tooltip);
    patch_W.addEventListener('onmouseout', hide_tooltip);

    var patch_X = svgDocument.getElementById('patch_23');
    patch_X.addEventListener('onmouseover', show_tooltip);
    patch_X.addEventListener('onmouseout', hide_tooltip);

    var patch_Y = svgDocument.getElementById('patch_24');
    patch_Y.addEventListener('onmouseover', show_tooltip);
    patch_Y.addEventListener('onmouseout', hide_tooltip);

    var patch_Z = svgDocument.getElementById('patch_25');
    patch_Z.addEventListener('onmouseover', show_tooltip);
    patch_Z.addEventListener('onmouseout', hide_tooltip);

    var tooltip = document.getElementById('tooltip_0');
    tooltip.style.visibility = "hidden";

    var tooltips = document.querySelectorAll("tooltip");
    for (var i = 0; i < tooltips.length; i++) {
        tooltips[i].addEventListener('onmouseover', show_tooltip);
        tooltips[i].addEventListener('onmouseout', hide_tooltip);
    }

    var patches = document.querySelectorAll("patch");
    for (var i = 0; i < patches.length; i++) {
        patches[i].addEventListener('onmouseover', show_tooltip);
        patches[i].addEventListener('onmouseout', hide_tooltip);
    }

]>>]
"""
function ShowTooltip(obj) {
    var cur = obj.id.slice(-1);
    var tip = svgDocument.getElementById('tooltip_' + cur);
    tip.setAttribute('visibility','visible')
}

function HideTooltip(obj) {
    var cur = obj.id.slice(-1);
    var tip = svgDocument.getElementById('tooltip_' + cur);
    tip.setAttribute('visibility','hidden')
}

]]>
</script>

""

# Insert the script at the top of the file and save it.
tree.insert(0, ET.XML(script))
ET.ElementTree(tree).write('svg_tooltip.svg')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

## 95.28 user_interfaces example code: wxcursor_demo.py

[source code]

""

Example to draw a cursor and report the data coords in wx

""

import matplotlib
matplotlib.use('WXAgg')

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from matplotlib.figure import Figure
from numpy import arange, sin, pi

import wx

class CanvasFrame(wx.Frame):
    def __init__(self,):
        wx.Frame.__init__(self,None,-1,'CanvasFrame',size=(550,350))
        self.SetBackgroundColour(wx.NamedColour("WHITE"))
self.figure = Figure()
self.axes = self.figure.add_subplot(111)
t = arange(0.0,3.0,0.01)
s = sin(2*pi*t)

self.axes.plot(t, s)
self.axes.set_xlabel('t')
self.axes.set_ylabel('sin(t)')
self.figure_canvas = FigureCanvas(self, -1, self.figure)

# Note that event is a MplEvent
self.figure_canvas.mpl_connect('motion_notify_event', self.UpdateStatusBar)
self.figure_canvas.Bind(wx.EVT_ENTER_WINDOW, self.ChangeCursor)

self.sizer = wx.BoxSizer(wx.VERTICAL)
self.sizer.Add(self.figure_canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
self.SetSizer(self.sizer)
self.Fit()

self.statusBar = wx.StatusBar(self, -1)
self.statusBar.SetFieldsCount(1)
self.SetStatusBar(self.statusBar)

self.toolbar = NavigationToolbar2Wx(self.figure_canvas)
self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
self.toolbar.Show()

def ChangeCursor(self, event):
    self.figure_canvas.SetCursor(wx.StockCursor(wx.CURSOR_BULLSEYE))

def UpdateStatusBar(self, event):
    if event.inaxes:
        x, y = event.xdata, event.ydata
        self.statusBar.SetStatusText(("x= " + str(x) + 
" y= " +str(y) ),
0)

class App(wx.App):
    def OnInit(self):
        'Create the main window and insert the custom frame'
        frame = CanvasFrame()
        self.SetTopWindow(frame)
        frame.Show(True)
        return True

if __name__ == '__main__':
    app = App(0)
    app.MainLoop()
96.1 widgets example code: buttons.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Button

freqs = np.arange(2, 20, 3)

fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.2)
t = np.arange(0.0, 1.0, 0.001)
s = np.sin(2*np.pi*freqs[0]*t)
l, = plt.plot(t, s, lw=2)

class Index:
    ind = 0
    def next(self, event):
        self.ind += 1
        i = self.ind % len(freqs)
ydata = np.sin(2*np.pi*freqs[i]*t)
l.set_ydata(ydata)
plt.draw()

    def prev(self, event):
        self.ind -= 1
        i = self.ind % len(freqs)
ydata = np.sin(2*np.pi*freqs[i]*t)
l.set_ydata(ydata)
plt.draw()

callback = Index()
axprev = plt.axes([0.7, 0.05, 0.1, 0.075])
axnext = plt.axes([0.81, 0.05, 0.1, 0.075])
bnext = Button(axnext, 'Next')
bnext.on_clicked(callback.next)
bprev = Button(axprev, 'Previous')
```

2513
bprev.on_clicked(callback.prev)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

## 96.2 widgets example code: check_buttons.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import CheckButtons

t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(6*np.pi*t)

fig, ax = plt.subplots()
l0, = ax.plot(t, s0, visible=False, lw=2)
l1, = ax.plot(t, s1, lw=2)
l2, = ax.plot(t, s2, lw=2)
plt.subplots_adjust(left=0.2)

rax = plt.axes([0.05, 0.4, 0.1, 0.15])
check = CheckButtons(rax, ('2 Hz', '4 Hz', '6 Hz'), (False, True, True))

def func(label):
    if label == '2 Hz': l0.set_visible(not l0.get_visible())
    elif label == '4 Hz': l1.set_visible(not l1.get_visible())
    elif label == '6 Hz': l2.set_visible(not l2.get_visible())
    plt.draw()
check.on_clicked(func)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

## 96.3 widgets example code: cursor.py

[source code]

```python
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt

from matplotlib.widgets import CheckButtons
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, axisbg='FFFFCC')

x, y = 4*(np.random.rand(2, 100)-.5)
ax.plot(x, y, 'o')
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

# set useblit = True on gtkagg for enhanced performance
cursor = Cursor(ax, useblit=True, color='red', linewidth=2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 96.4 widgets example code: `lasso_selector_demo.py`

[source code]

```python
from __future__ import print_function
import numpy as np

from matplotlib.widgets import LassoSelector
from matplotlib.path import Path

try:
    raw_input
except NameError:
    # Python 3
    raw_input = input

class SelectFromCollection(object):
    """Select indices from a matplotlib collection using 'LassoSelector'.

    Selected indices are saved in the 'ind' attribute. This tool highlights
    selected points by fading them out (i.e., reducing their alpha values).
    If your collection has alpha < 1, this tool will permanently alter them.

    Note that this tool selects collection objects based on their *origins*
    (i.e., 'offsets').

    Parameters
    ----------
    ax : :class:`~matplotlib.axes.Axes`
        Axes to interact with.

    collection : :class:`~matplotlib.collections.Collection` subclass
    """
```

96.4. widgets example code: `lasso_selector_demo.py` 2515
To highlight a selection, this tool sets all selected points to an alpha value of 1 and non-selected points to `alpha_other`.

```python
def __init__(self, ax, collection, alpha_other=0.3):
    self.canvas = ax.figure.canvas
    self.collection = collection
    self.alpha_other = alpha_other

    self.xys = collection.get_offsets()
    self.Npts = len(self.xys)

    self.fc = collection.get_facecolors()
    if len(self.fc) == 0:
        raise ValueError('Collection must have a facecolor')
    elif len(self.fc) == 1:

    self.lasso = LassoSelector(ax, onselect=self.onselect)
    self.ind = []

def onselect(self, verts):
    path = Path(verts)
    self.ind = np.nonzero([path.contains_point(xy) for xy in self.xys])[0]
    self.fc[:, -1] = self.alpha_other
    self.fc[self.ind, -1] = 1
    self.collection.set_facecolors(self.fc)
    self.canvas.draw_idle()

def disconnect(self):
    self.lasso.disconnect_events()
    self.fc[:, -1] = 1
    self.collection.set_facecolors(self.fc)
    self.canvas.draw_idle()
```

```python
if __name__ == '__main__':
    import matplotlib.pyplot as plt

    plt.ion()
    data = np.random.rand(100, 2)

    subplot_kw = dict(xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
    fig, ax = plt.subplots(subplot_kw=subplot_kw)

    pts = ax.scatter(data[:, 0], data[:, 1], s=80)
    selector = SelectFromCollection(ax, pts)

    plt.draw()
    raw_input('Press any key to accept selected points')
```
print("Selected points:")
print(selector.xys[selector.ind])
selector.disconnect()

# Block end of script so you can check that the lasso is disconnected.
raw_input('Press any key to quit')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.5 widgets example code: menu.py

[source code]

from __future__ import division, print_function
import numpy as np
import matplotlib
import matplotlib.colors as colors
import matplotlib.patches as patches
import matplotlib.mathtext as mathtext
import matplotlib.pyplot as plt
import matplotlib.artist as artist
import matplotlib.image as image

class ItemProperties:
    def __init__(self, fontsize=14, labelcolor='black', bgcolor='yellow',
                 alpha=1.0):
        self.fontsize = fontsize
        self.labelcolor = labelcolor
        selfbgcolor = bgcolor
        self.alpha = alpha

        self.labelcolor_rgb = colors.colorConverter.to_rgb(labelcolor)
        self.bgcolor_rgb = colors.colorConverter.to_rgb(bgcolor)

class MenuItem(artist.Artist):
    parser = mathtext.MathTextParser("Bitmap")
    padx = 5
    pady = 5
    def __init__(self, fig, labelstr, props=None, hoverprops=None,
                 on_select=None):
        artist.Artist.__init__(self)

        self.set_figure(fig)
        self.labelstr = labelstr

        if props is None:
            props = ItemProperties()

        if hoverprops is None:
            hoverprops = ItemProperties()
self.props = props
self.hoverprops = hoverprops

self.on_select = on_select

x, self.depth = self.parser.to_mask(labelstr, fontsize=props.fontsize, dpi=fig.dpi)

if props.fontsize!=hoverprops.fontsize:
    raise NotImplementedError('support for different font sizes not implemented')

self.labelwidth = x.shape[1]
self.labelheight = x.shape[0]

self.labelArray = np.zeros((x.shape[0], x.shape[1], 4))
self.labelArray[:,:, -1] = x/255.

self.label = image.FigureImage(fig, origin='upper')
self.label.set_array(self.labelArray)

# we'll update these later
self.rect = patches.Rectangle((0,0), 1,1)
self.set_hover_props(False)

fig.canvas.mpl_connect('button_release_event', self.check_select)

def check_select(self, event):
    over, junk = self.rect.contains(event)
    if not over:
        return

    if self.on_select is not None:
        self.on_select(self)

def set_extent(self, x, y, w, h):
    print(x, y, w, h)
    self.rect.set_x(x)
    self.rect.set_y(y)
    self.rect.set_width(w)
    self.rect.set_height(h)

    self.label.ox = x+self.padx
    self.label.oy = y-self.depth+self.pady/2.

    self.rect._update_patch_transform()
    self.hover = False

def draw(self, renderer):
    self.rect.draw(renderer)
def set_hover_props(self, b):
    if b:
        props = self.hoverprops
    else:
        props = self.props

    r, g, b = props.labelcolor_rgb
    self.labelArray[:, :, 0] = r
    self.labelArray[:, :, 1] = g
    self.labelArray[:, :, 2] = b
    self.label.set_array(self.labelArray)
    self.rect.set(facecolor=props.bgcolor, alpha=props.alpha)

def set_hover(self, event):
    'check the hover status of event and return true if status is changed'
    b, junk = self.rect.contains(event)

    changed = (b != self.hover)

    if changed:
        self.set_hover_props(b)

    self.hover = b
    return changed

class Menu:

def __init__(self, fig, menuitems):
    self.figure = fig
    fig.suppressComposite = True

    self.menuitems = menuitems
    self.numitems = len(menuitems)

    maxw = max([item.labelwidth for item in menuitems])
    maxh = max([item.labelheight for item in menuitems])

    totalh = self.numitems*maxh + (self.numitems+1)*2*MenuItem.pady

    x0 = 100
    y0 = 400

    width = maxw + 2*MenuItem.padx
    height = maxh+MenuItem.pady

    for item in menuitems:
        left = x0
        bottom = y0-maxh-MenuItem.pady
item.set_extent(left, bottom, width, height)
fig.artists.append(item)
y0 -= maxh + MenuItem.pady

fig.canvas.mpl_connect('motion_notify_event', self.on_move)

def on_move(self, event):
    draw = False
    for item in self.menuitems:
        draw = item.set_hover(event)
        if draw:
            self.figure.canvas.draw()
            break

fig = plt.figure()
fig.subplots_adjust(left=0.3)
props = ItemProperties(labelcolor='black', bgcolor='yellow',
                       fontsize=15, alpha=0.2)
hoverprops = ItemProperties(labelcolor='white', bgcolor='blue',
                           fontsize=15, alpha=0.2)

menuitems = []
for label in ('open', 'close', 'save', 'save as', 'quit'):
    def on_select(item):
        print('you selected %s' % item.labelstr)
        item = MenuItem(fig, label, props=props, hoverprops=hoverprops,
                        on_select=on_select)
        menuitems.append(item)

menu = Menu(fig, menuitems)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.6 widgets example code: multicursor.py

[source code]

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import MultiCursor

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.plot(t, s1)

ax2 = fig.add_subplot(212, sharex=ax1)
ax2.plot(t, s2)

multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.7 widgets example code: radio_buttons.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import RadioButtons

t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(8*np.pi*t)

fig, ax = plt.subplots()
l, = ax.plot(t, s0, lw=2, color='red')
plt.subplots_adjust(left=0.3)
axcolor = 'lightgoldenrodyellow'
rax = plt.axes([0.05, 0.7, 0.15, 0.15], axisbg=axcolor)
radio = RadioButtons(rax, ('2 Hz', '4 Hz', '8 Hz'))

def hzfunc(label):
    hzdict = {'2 Hz':s0, '4 Hz':s1, '8 Hz':s2}
ydata = hzdict[label]
l.set_ydata(ydata)
plt.draw()
radio.on_clicked(hzfunc)

rax = plt.axes([0.05, 0.4, 0.15, 0.15], axisbg=axcolor)
radio2 = RadioButtons(rax, ('red', 'blue', 'green'))

def colorfunc(label):
    l.set_color(label)
    plt.draw()
radio2.on_clicked(colorfunc)

rax = plt.axes([0.05, 0.1, 0.15, 0.15], axisbg=axcolor)
radio3 = RadioButtons(rax, ('-', '--', '-.', 'steps', ':'))

def stylefunc(label):
    l.set_linestyle(label)
    plt.draw()
```
from __future__ import print_function

Do a mouseclick somewhere, move the mouse to some destination, release
the button. This class gives click- and release-events and also draws
a line or a box from the click-point to the actual mouseposition
(within the same axes) until the button is released. Within the
method 'self.ignore()' it is checked whether the button from eventpress
and eventrelease are the same.

from matplotlib.widgets import RectangleSelector
import numpy as np
import matplotlib.pyplot as plt

def line_select_callback(eclick, erelease):
    'eclick and erelease are the press and release events'
    x1, y1 = eclick.xdata, eclick.ydata
    x2, y2 = erelease.xdata, erelease.ydata
    print ('(%3.2f, %3.2f) --> (%3.2f, %3.2f)' % (x1, y1, x2, y2))
    print ('The button you used were: %s %s' % (eclick.button, erelease.button))

def toggle_selector(event):
    print ('Key pressed.
    if event.key in ['Q', 'q'] and toggle_selector.RS.active:
        print ('RectangleSelector deactivated.')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.RS.active:
        print ('RectangleSelector activated.')
        toggle_selector.RS.set_active(True)

fig, current_ax = plt.subplots() # make a new plottingrange
N = 100000 # If N is large one can see
x = np.linspace(0.0, 10.0, N) # improvement by use blitting!

plt.plot(x, +np.sin(.2*np.pi*x), lw=3.5, c='b', alpha=.7) # plot something
plt.plot(x, +np.cos(.2*np.pi*x), lw=3.5, c='r', alpha=.5)
plt.plot(x, -np.sin(.2*np.pi*x), lw=3.5, c='g', alpha=.3)

print ('\n click --> release')
# drawtype is 'box' or 'line' or 'none'
toggle_selector.RS = RectangleSelector(current_ax, line_select_callback,
    drawtype='box', useblit=True,
    button=[1,3], # don't use middle button
    minspanx=5, minspany=5,
    spancoords='pixels')

plt.connect('key_press_event', toggle_selector)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.9 widgets example code: slider_demo.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider, Button, RadioButtons

fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25)
t = np.arange(0.0, 1.0, 0.001)
a0 = 5
f0 = 3
s = a0*np.sin(2*np.pi*f0*t)
l, = plt.plot(t,s, lw=2, color='red')
plt.axis([0, 1, -10, 10])

axcolor = 'lightgoldenrodyellow'
axfreq = plt.axes([0.25, 0.1, 0.65, 0.03], axisbg=axcolor)
axamp = plt.axes([0.25, 0.15, 0.65, 0.03], axisbg=axcolor)
sfreq = Slider(axfreq, 'Freq', 0.1, 30.0, valinit=f0)
samp = Slider(axamp, 'Amp', 0.1, 10.0, valinit=a0)

def update(val):
    amp = samp.val
    freq = sfreq.val
    l.set_ydata(amp*np.sin(2*np.pi*freq*t))
    fig.canvas.draw_idle()
    sfreq.on_changed(update)
    samp.on_changed(update)

resetax = plt.axes([0.8, 0.025, 0.15, 0.04])
button = Button(resetax, 'Reset', color=axcolor, hovercolor='0.975')

def reset(event):
    sfreq.reset()
    samp.reset()
    button.on_clicked(reset)

rax = plt.axes([0.025, 0.5, 0.15, 0.15], axisbg=axcolor)
```

96.9. widgets example code: slider_demo.py 2523
```python
radio = RadioButtons(rax, ('red', 'blue', 'green'), active=0)
def colorfunc(label):
    l.set_color(label)
    fig.canvas.draw_idle()
radio.on_clicked(colorfunc)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 96.10 widgets example code: span_selector.py

[source code]

```python
#!/usr/bin/env python

""
The SpanSelector is a mouse widget to select a xmin/xmax range and plot the
detail view of the selected region in the lower axes
""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import SpanSelector

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(211, axisbg='#FFFFCC')

x = np.arange(0.0, 5.0, 0.01)
y = np.sin(2*np.pi*x) + 0.5*np.random.randn(len(x))

ax.plot(x, y, '-')
ax.set_ylim(-2,2)
ax.set_title('Press left mouse button and drag to test')

ax2 = fig.add_subplot(212, axisbg='#FFFFCC')
line2, = ax2.plot(x, y, '-')

def onselect(xmin, xmax):
    indmin, indmax = np.searchsorted(x, (xmin, xmax))
    indmax = min(len(x)-1, indmax)
    thisx = x[indmin:indmax]
    thisy = y[indmin:indmax]
    line2.set_data(thisx, thisy)
    ax2.set_xlim(thisx[0], thisx[-1])
    ax2.set_ylim(thisy.min(), thisy.max())
    fig.canvas.draw()

# set useblit True on gtkagg for enhanced performance
span = SpanSelector(ax, onselect, 'horizontal', useblit=True,
                       rectprops=dict(alpha=0.5, facecolor='red'))
```

2524 Chapter 96. widgets Examples
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)
Part X

Glossary
AGG  The Anti-Grain Geometry (Agg) rendering engine, capable of rendering high-quality images

Cairo  The Cairo graphics engine

dateutil  The dateutil library provides extensions to the standard datetime module

EPS  Encapsulated Postscript (EPS)

freetype  freetype is a font rasterization library used by matplotlib which supports TrueType, Type 1, and OpenType fonts.

GDK  The Gimp Drawing Kit for GTK+

GTK  The GIMP Toolkit (GTK) graphical user interface library

JPG  The Joint Photographic Experts Group (JPEG) compression method and file format for photographic images

numpy  numpy is the standard numerical array library for python, the successor to Numeric and numarray. numpy provides fast operations for homogeneous data sets and common mathematical operations like correlations, standard deviation, fourier transforms, and convolutions.

PDF  Adobe’s Portable Document Format (PDF)

PNG  Portable Network Graphics (PNG), a raster graphics format that employs lossless data compression which is more suitable for line art than the lossy jpg format. Unlike the gif format, png is not encumbered by requirements for a patent license.

PS  Postscript (PS) is a vector graphics ASCII text language widely used in printers and publishing. Postscript was developed by adobe systems and is starting to show its age: for example is does not have an alpha channel. PDF was designed in part as a next-generation document format to replace postscript

pygtk  pygtk provides python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend. Widely used on linux, and is often packages as ‘python-gtk2’

pyqt  pyqt provides python wrappers for the Qt widgets library and is required by the matplotlib QtAgg and Qt4Agg backends. Widely used on linux and windows; many linux distributions package this as ‘python-qt3’ or ‘python-qt4’.

python  python is an object oriented interpreted language widely used for scripting, application development, web application servers, scientific computing and more.

pytz  pytz provides the Olson tz database in Python. it allows accurate and cross platform timezone calculations and solves the issue of ambiguous times at the end of daylight savings

Qt  Qt is a cross-platform application framework for desktop and embedded development.

Qt4  Qt4 is the most recent version of Qt cross-platform application framework for desktop and embedded development.

raster graphics  Raster graphics, or bitmaps, represent an image as an array of pixels which is resolution dependent. Raster graphics are generally most practical for photo-realistic images, but do not scale easily without loss of quality.

SVG  The Scalable Vector Graphics format (SVG). An XML based vector graphics format supported by many web browsers.
TIFF  Tagged Image File Format (TIFF) is a file format for storing images, including photographs and line art.

Tk  Tk is a graphical user interface for Tcl and many other dynamic languages. It can produce rich, native applications that run unchanged across Windows, Mac OS X, Linux and more.

vector graphics  vector graphics use geometrical primitives based upon mathematical equations to represent images in computer graphics. Primitives can include points, lines, curves, and shapes or polygons. Vector graphics are scalable, which means that they can be resized without suffering from issues related to inherent resolution like are seen in raster graphics. Vector graphics are generally most practical for typesetting and graphic design applications.

wxpython  wxpython provides python wrappers for the wxWidgets library for use with the WX and WX-Agg backends. Widely used on linux, OS-X and windows, it is often packaged by linux distributions as ‘python-wxgtk’

wxWidgets  WX is cross-platform GUI and tools library for GTK, MS Windows, and MacOS. It uses native widgets for each operating system, so applications will have the look-and-feel that users on that operating system expect.
BIBLIOGRAPHY

[list-colormaps]  https://gist.github.com/endolith/2719900#id7

2531
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplotlib.afm</td>
<td>743</td>
</tr>
<tr>
<td>matplotlib.animation</td>
<td>747</td>
</tr>
<tr>
<td>matplotlib.artist</td>
<td>757</td>
</tr>
<tr>
<td>matplotlib.axis</td>
<td>953</td>
</tr>
<tr>
<td>matplotlib.backend_bases</td>
<td>963</td>
</tr>
<tr>
<td>matplotlib.backends.backend_pdf</td>
<td>983</td>
</tr>
<tr>
<td>matplotlib.backends.backend_qt4agg</td>
<td>981</td>
</tr>
<tr>
<td>matplotlib.backends.backend_wxagg</td>
<td>982</td>
</tr>
<tr>
<td>matplotlib.cbook</td>
<td>987</td>
</tr>
<tr>
<td>matplotlib.cm</td>
<td>1001</td>
</tr>
<tr>
<td>matplotlib.collections</td>
<td>1005</td>
</tr>
<tr>
<td>matplotlib.colorbar</td>
<td>1159</td>
</tr>
<tr>
<td>matplotlib.colors</td>
<td>1165</td>
</tr>
<tr>
<td>matplotlib.dates</td>
<td>1175</td>
</tr>
<tr>
<td>matplotlib.dviread</td>
<td>1187</td>
</tr>
<tr>
<td>matplotlib.figure</td>
<td>1191</td>
</tr>
<tr>
<td>matplotlib.finance</td>
<td>1213</td>
</tr>
<tr>
<td>matplotlib.font_manager</td>
<td>1225</td>
</tr>
<tr>
<td>matplotlib.fontconfig_pattern</td>
<td>1231</td>
</tr>
<tr>
<td>matplotlib.gridspec</td>
<td>1233</td>
</tr>
<tr>
<td>matplotlib.image</td>
<td>1237</td>
</tr>
<tr>
<td>matplotlib.legend</td>
<td>1243</td>
</tr>
<tr>
<td>matplotlib.legend_handler</td>
<td>1246</td>
</tr>
<tr>
<td>matplotlib.lines</td>
<td>1251</td>
</tr>
<tr>
<td>matplotlib.markers</td>
<td>1261</td>
</tr>
<tr>
<td>matplotlib.mathtext</td>
<td>1267</td>
</tr>
<tr>
<td>matplotlib.mlab</td>
<td>1283</td>
</tr>
<tr>
<td>matplotlib.offsetbox</td>
<td>1315</td>
</tr>
<tr>
<td>matplotlib.patches</td>
<td>1327</td>
</tr>
<tr>
<td>matplotlib.path</td>
<td>1365</td>
</tr>
<tr>
<td>matplotlib.pathheffects</td>
<td>199</td>
</tr>
<tr>
<td>matplotlib.projections</td>
<td>469</td>
</tr>
<tr>
<td>matplotlib.projections.polar</td>
<td>469</td>
</tr>
<tr>
<td>matplotlib.pyplot</td>
<td>1377</td>
</tr>
<tr>
<td>matplotlib.sankey</td>
<td>1559</td>
</tr>
<tr>
<td>matplotlib.scale</td>
<td>467</td>
</tr>
<tr>
<td>matplotlib.sphinxext.plot_directive</td>
<td>431</td>
</tr>
<tr>
<td>matplotlib.spines</td>
<td>1567</td>
</tr>
<tr>
<td>matplotlib.style</td>
<td>1571</td>
</tr>
<tr>
<td>matplotlib.text</td>
<td>1573</td>
</tr>
<tr>
<td>matplotlib.ticker</td>
<td>1587</td>
</tr>
<tr>
<td>matplotlib.tight_layout</td>
<td>1597</td>
</tr>
<tr>
<td>matplotlib.transforms</td>
<td>443</td>
</tr>
<tr>
<td>matplotlib.tri</td>
<td>1599</td>
</tr>
<tr>
<td>matplotlib.typelfont</td>
<td>1611</td>
</tr>
<tr>
<td>matplotlib.units</td>
<td>1613</td>
</tr>
<tr>
<td>matplotlib.widgets</td>
<td>1615</td>
</tr>
<tr>
<td>mpl_toolkits.axes_grid.axes_size</td>
<td>679</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.art3d</td>
<td>635</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.axes3d</td>
<td>614</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.axis3d</td>
<td>633</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.proj3d</td>
<td>639</td>
</tr>
</tbody>
</table>
Symbols
__call__() (matplotlib.tri.CubicTriInterpolator method), 1603
__call__() (matplotlib.tri.LinearTriInterpolator method), 1601
__call__() (matplotlib.tri.TrapezoidMapTriFinder method), 1600

A
AbstractPathEffect (class in matplotlib.path_effects), 1373
Accent (class in matplotlib.mathtext), 1267
accent() (matplotlib.mathtext.Parser method), 1276
acorr() (in module matplotlib.pyplot), 1377
acorr() (matplotlib.axes.Axes method), 767
add() (matplotlib.figure.AxesStack method), 1191
add() (matplotlib.mlab.FIFOBuff method), 1285
add() (matplotlib.sankey.Sankey method), 1563
add_artist() (matplotlib.axes.Axes method), 768
add_artist() (matplotlib.offsetbox.AuxTransformBox method), 1318
add_artist() (matplotlib.offsetbox.DrawingArea method), 1320
add_auto_adjustable_area()
 (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
add_axes() (matplotlib.figure.Figure method), 1192
add_axobserver() (matplotlib.figure.Figure method), 1194
add_callback() (matplotlib.artist.Artist method), 757
add_callback() (matplotlib.axes.Axes method), 768
add_callback()
 (matplotlib.backend_bases.TimerBase method), 980
add_callback() (matplotlib.collections.AsteriskPolygonCollection method), 1006
add_callback() (matplotlib.collections.BrokenBarHCollection method), 1017
add_callback() (matplotlib.collections.CircleCollection method), 1027
add_callback() (matplotlib.collections.Collection method), 1038
add_callback() (matplotlib.collections.EllipseCollection method), 1049
add_callback() (matplotlib.collections.EventCollection method), 1060
add_callback() (matplotlib.collections.LineCollection method), 1072
add_callback() (matplotlib.collections.PatchCollection method), 1083
add_callback() (matplotlib.collections.PathCollection method), 1093
add_callback() (matplotlib.collections.PolyCollection method), 1104
add_callback() (matplotlib.collections.QuadMesh method), 1114
add_callback() (matplotlib.collections.RegularPolyCollection method), 1125
add_callback() (matplotlib.collections.StarPolygonCollection method), 1136
add_callback() (matplotlib.collections.TriMesh method), 1147
add_checker() (matplotlib.cm.ScalarMappable method), 2535
add_checker() (matplotlib.collections.AsteriskPolygonCollection
method), 1006
add_checker() (matplotlib.collections.BrokenBarHCollection
method), 1017
add_checker() (matplotlib.collections.CircleCollection
method), 1028
add_checker() (matplotlib.collections.Collection
method), 1039
add_checker() (matplotlib.collections.EllipseCollection
method), 1049
add_checker() (matplotlib.collections.EventCollection
method), 1060
add_checker() (matplotlib.collections.LineCollection
method), 1072
add_checker() (matplotlib.collections.PatchCollection
method), 1083
add_checker() (matplotlib.collections.PathCollection
method), 1093
add_checker() (matplotlib.collections.QuadMesh
method), 1114
add_checker() (matplotlib.collections.RegularPolyCollection
method), 1125
add_checker() (matplotlib.collections.StarPolygonCollection
method), 1136
add_checker() (matplotlib.collections.TriMesh
method), 1147
add_collection() (matplotlib.axes.Axes method), 768
add_collection3d() (mpl_toolkits.mplot3d.Axes3D
method), 547, 607
add_collection3d() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 555, 614
add_contour_set() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 555, 614
add_contourf_set() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 555, 614
add_line() (matplotlib.axes.Axes method), 769
add_lines() (matplotlib.collections.AsteriskPolygonCollection
method), 1006
add_lines() (matplotlib.collections.BrokenBarHCollection
method), 1017
add_lines() (matplotlib.collections.CircleCollection
method), 1028
add_lines() (matplotlib.collections.Collection
method), 1039
add_lines() (matplotlib.collections.EllipseCollection
method), 1049
add_lines() (matplotlib.collections.EventCollection
method), 1060
add_lines() (matplotlib.collections.LineCollection
method), 1072
add_lines() (matplotlib.collections.PatchCollection
method), 1083
add_lines() (matplotlib.collections.PathCollection
method), 1093
add_lines() (matplotlib.collections.QuadMesh
method), 1114
add_lines() (matplotlib.collections.RegularPolyCollection
method), 1125
add_lines() (matplotlib.collections.StarPolygonCollection
method), 1136
add_lines() (matplotlib.collections.TriMesh
method), 1147
add_line() (matplotlib.axes.Axes method), 769
add_lines() (matplotlib.colorbar.Colorbar
method), 1159
add_lines() (matplotlib.colorbar.ColorbarBase
method), 1160
add_patch() (matplotlib.axes.Axes method), 769
add_positions() (matplotlib.collections.EventCollection
method), 1060
add_subplot() (matplotlib.figure.Figure method), 1194
add_table() (matplotlib.axes.Axes method), 769
adjust_drawing_area() (matplotlib.legend_handler.HandlerBase
method), 1247
align_iterators() (in module matplotlib.cbook), 991
allequal() (in module matplotlib.cbook), 992
allow_rasterization() (in module matplotlib.artist), 764
allowed_metadata (matplotlib.animation.MencoderBase attribute), 751
allpairs() (in module matplotlib.mlab), 1289
alltrue() (in module matplotlib.cbook), 992
align_iterators() (in module matplotlib.cbook), 991
allequal() (in module matplotlib.cbook), 992
allow_rasterization() (in module matplotlib.artist), 764
align_iterators() (in module matplotlib.mlab), 1289
allequal() (in module matplotlib.cbook), 992
allow_rasterization() (in module matplotlib.artist), 764
Index 2537

append_axes() (mpl_toolkits.axes_grid.axes_divider.AxesDivider method), 523, 683
append_positions() (matplotlib.collections.EventCollection method), 1060
append_size() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
apply_aspect() (matplotlib.axes.Axes method), 776
apply_tickdir() (matplotlib.axis.Tick method), 958
apply_tickdir() (matplotlib.axis.XTick method), 960
apply_tickdir() (matplotlib.axis.YTick method), 961
apply_window() (in module matplotlib.mlab), 1289
Arc (class in matplotlib.patches), 1327
arc() (matplotlib.path.Path class method), 1366
args_key (matplotlib.animation.AVConvBase attribute), 747
args_key (matplotlib.animation.FFMpegBase attribute), 749
args_key (matplotlib.animation.ImageMagickBase attribute), 750
args_key (matplotlib.animation.MencoderBase attribute), 751
Arrow (class in matplotlib.patches), 1328
arrow() (in module matplotlib.pyplot), 1385
arrow() (matplotlib.axes.Axes method), 776
ArrowStyle (class in matplotlib.patches), 1329
ArrowStyle.BarAB (class in matplotlib.patches), 1331
ArrowStyle.BracketA (class in matplotlib.patches), 1331
ArrowStyle.BracketAB (class in matplotlib.patches), 1331
ArrowStyle.BracketB (class in matplotlib.patches), 1332
ArrowStyle.Curve (class in matplotlib.patches), 1332
ArrowStyle.CurveA (class in matplotlib.patches), 1332
ArrowStyle.CurveAB (class in matplotlib.patches), 1332
ArrowStyle.CurveB (class in matplotlib.patches), 1332
ArrowStyle.CurveFilledA (class in matplotlib.patches), 1332
ArrowStyle.CurveFilledAB (class in matplotlib.patches), 1332
ArrowStyle.CurveFilledB (class in matplotlib.patches), 1333
autoscale_None() (matplotlib.collections.EventCollection method), 1060
autoscale_None() (matplotlib.collections.LineCollection method), 1072
autoscale_None() (matplotlib.collections.PatchCollection method), 1083
autoscale_None() (matplotlib.collections.PathCollection method), 1093
autoscale_None() (matplotlib.collections.PolyCollection method), 1104
autoscale_None() (matplotlib.collections.QuadMesh method), 1114
autoscale_None() (matplotlib.collections.RegularPolyCollection method), 1125
autoscale_None() (matplotlib.collections.StarPolygonCollection method), 1136
autoscale_None() (matplotlib.collections.TriMesh method), 1147
autoscale_None() (matplotlib.colors.LogNorm method), 1170
autoscale_None() (matplotlib.colors.Normalize method), 1171
autoscale_None() (matplotlib.colors.PowerNorm method), 1171
autoscale_None() (matplotlib.colors.SymLogNorm method), 1172
autoscale_view() (matplotlib.axes.Axes method), 778
autoscale_view() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 778
ax (matplotlib.colorbar.ColorbarBase attribute),
A VConvBase (class in matplotlib.animation), 747
A VConvFileWriter (class in matplotlib.animation), 747
A VConvWriter (class in matplotlib.animation), 747
AutOWidthChar (class in matplotlib.mathtext), 1267
AutOWidthChar (class in matplotlib.mathtext), 1267
AuxTransformBox (class in matplotlib.offsetbox), 1317
available() (matplotlib.widgets.LockDraw method), 1388
available() (matplotlib.widgets.LockDraw method), 1388
AVConvBase (class in matplotlib.animation), 747
AVConvFileWriter (class in matplotlib.animation), 747
AVConvWriter (class in matplotlib.animation), 747
Axis (class in matplotlib.axis), 953
Axis (class in matplotlib.axis), 953
AxisArtist (class in mpl_toolkits.axes_grid.axis_artist), 525
AxisArtist (class in mpl_toolkits.axes_grid.axis_artist), 525
axes (matplotlib.figure.Figure attribute), 1196
axes() (in module matplotlib.pyplot), 1388
Axes (class in matplotlib.axes), 767
Axes (class in matplotlib.axes), 767
Axes3D (class in mpl_toolkits.mplot3d.axes3d), 555
AxesStack (class in matplotlib.figure), 1191
AxesWidget (class in matplotlib.widgets), 1615
auxhline() (in module matplotlib.pyplot), 1388
auxhline() (matplotlib.axes.Axes method), 778
axhspan() (in module matplotlib.pyplot), 1390
axhspan() (matplotlib.axes.Axes method), 780
Axis (class in matplotlib.axis), 953
Axis (class in matplotlib.axis), 953
AxisLabel (class in mpl_toolkits.axes_grid.axis_artist), 525
AxisLabel (class in mpl_toolkits.axes_grid.axis_artist), 525
AxisInfo (class in matplotlib.units), 1613
AxisInfo (class in matplotlib.units), 1613
axis() (matplotlib.axes.Axes method), 782
axis() (matplotlib.axes.Axes method), 782
axis_date() (matplotlib.axis.Axes method), 953
axis_name (matplotlib.axis.XAxis attribute), 959
axis_name (matplotlib.axis.YAxis attribute), 960
AxisArtist (class in mpl_toolkits.axes_grid.axis_artist), 525
AxisArtist (class in mpl_toolkits.axes_grid.axis_artist), 525
AxisInfo (class in matplotlib.units), 1613
AxisInfo (class in matplotlib.units), 1613
axisinfo() (matplotlib.units.ConversionInterface static method), 1614
axisinfo() (matplotlib.units.ConversionInterface static method), 1614
back() (matplotlib.backend_bases.NavigationToolbar2
back() (matplotlib.cbook.Stack method), 990
BakomaFonts (class in matplotlib.mathtext), 1267
bar() (module matplotlib.pyplot), 1396
bar() (matplotlib.axes.Axes method), 785
bar() (mpl_toolkits.mplot3d.Axes3D method), 548, 608
bar3d() (mpl_toolkits.mplot3d.Axes3D method), 556, 615
barbs() (module matplotlib.pyplot), 1399
barbs() (matplotlib.axes.Axes method), 788
barh() (module matplotlib.pyplot), 1403
barh() (matplotlib.axes.Axes method), 792
base() (matplotlib.ticker.LogFormatter method), 1591
base() (matplotlib.ticker.LogLocator method), 1594
base_repr() (module matplotlib.mlab), 1290
bboxpath (matplotlib.mathtext.StandardPsFonts attribute), 1278
Bbox (class in matplotlib.transforms), 448
bbox_artist() (module matplotlib.offsetbox), 1325
bbox_artist() (module matplotlib.transforms), 1364
BboxBase (class in matplotlib.transforms), 445
BboxImage (class in matplotlib.image), 1237
BboxTransform (class in matplotlib.transforms), 462
BboxTransformFrom (class in matplotlib.transforms), 463
BboxTransformTo (class in matplotlib.transforms), 463
bin_boundaries() (matplotlib.ticker.MaxNLocator method), 1594
bin_path() (matplotlib.animation.MovieWriter class method), 752
binary_repr() (module matplotlib.mlab), 1290
binom() (matplotlib.mathtext.Parser method), 1276
bivariater_normal() (module matplotlib.mlab), 1290
blended_transform_factory() (module matplotlib.transforms), 461
BlendedAffine2D (class in matplotlib.transforms), 460
BlendedGenericTransform (class in matplotlib.transforms), 460
blit() (matplotlib.backends.backend_wxagg.FigureCanvasWxAgg method), 982
bone() (module matplotlib.pyplot), 1405
BoundaryNorm (class in matplotlib.colors), 1166
bounds (matplotlib.transforms.BboxBase attribute), 445
Box (class in matplotlib.mathtext), 1268
box() (module matplotlib.pyplot), 1405
boxplot() (module matplotlib.pyplot), 1405
boxplot() (matplotlib.axes.Axes method), 794
boxplot_stats() (module matplotlib.cbook), 992
BoxStyle (class in matplotlib.patches), 1333
BoxStyle.Circle (class in matplotlib.patches), 1335
BoxStyle.LArrow (class in matplotlib.patches), 1335
BoxStyle.RArrow (class in matplotlib.patches), 1335
BoxStyle.Round (class in matplotlib.patches), 1336
BoxStyle.Round4 (class in matplotlib.patches), 1336
BoxStyle.Roundtooth (class in matplotlib.patches), 1336
BoxStyle.Sawtooth (class in matplotlib.patches), 1336
BoxStyle.Square (class in matplotlib.patches), 1336
broken_barh() (module matplotlib.pyplot), 1409
broken_barh() (matplotlib.axes.Axes method), 798
BrokenBarHCollection (class in matplotlib.collections), 1016
bubble() (matplotlib.cbook.Stack method), 991
bubble() (matplotlib.figure.AxesStack method), 1191
Bunch (class in matplotlib.cbook), 987
Button (class in matplotlib.widgets), 1615
button (matplotlib.backend_bases.MouseEvent attribute), 974
button_press_event() (matplotlib.backend_bases.MouseEvent method), 964
button_release_event() (matplotlib.backend_bases.MouseEvent method), 964
bxp() (matplotlib.axes.Axes method), 800
byAttribute() (matplotlib.cbook.Sorter method), 990
byItem() (matplotlib.cbook.Sorter method), 990
C
c_over_c() (matplotlib.mathtext.Parser method), 1276
cache (matplotlib.colors.ColorConverter attribute), 1166
Cairo, 2529
calculate_plane_coefficients() (matplotlib.tri.Triangulation method), 1599
CallbackRegistry (class in matplotlib.cbook), 987
can_pan() (matplotlib.axes.Axes method), 804
can_pan() (matplotlib.projections.polar.PolarAxes method), 472
can_pan() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 556, 615
can_zoom() (matplotlib.axes.Axes method), 804
can_zoom() (matplotlib.projections.polar.PolarAxes method), 472
can_zoom() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 556, 615
candlestick() (in module matplotlib.finance), 1213
candlestick2() (in module matplotlib.finance), 1213
candlestick2_ochl() (in module matplotlib.finance), 1214
candlestick2_ohlc() (in module matplotlib.finance), 1214
candlestick_ochl() (in module matplotlib.finance), 1215
candlestick_ohlc() (in module matplotlib.finance), 1215
center() (matplotlib.mlab.PCA method), 1288
center_matrix() (in module matplotlib.mlab), 1290
change_geometry() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 522, 682
changed() (matplotlib.cm.ScalarMappable method), 1001
changed() (matplotlib.collections.AsteriskPolygonCollection method), 1006
changed() (matplotlib.collections.BrokenBarHCollection method), 1017
changed() (matplotlib.collections.CircleCollection method), 1028
changed() (matplotlib.collections.Collection method), 1039
changed() (matplotlib.collections.EllipseCollection method), 1049
changed() (matplotlib.collections.EventCollection method), 1060
changed() (matplotlib.collections.LineCollection method), 1072
changed() (matplotlib.collections.PatchCollection method), 1083
changed() (matplotlib.collections.PathCollection method), 1093
changed() (matplotlib.collections.PolyCollection method), 1104
changed() (matplotlib.collections.QuadMesh method), 1115
changed() (matplotlib.collections.RegularPolyCollection method), 1125
changed() (matplotlib.collections.StarPolygonCollection method), 1136
changed() (matplotlib.collections.TriangleCollection method), 1147
changed() (matplotlib.image.PcolorImage method), 1239
Char (class in matplotlib.mathtext), 1268
check_update() (matplotlib.cm.ScalarMappable method), 1001
check_update() (matplotlib.collections.AsteriskPolygonCollection method), 1006
check_update() (matplotlib.collections.BrokenBarHCollection method), 1017
check_update() (matplotlib.collections.CircleCollection method), 1028
check_update() (matplotlib.collections.Collection method), 1039
check_update() (matplotlib.collections.EllipseCollection method), 1049
check_update() (matplotlib.collections.EventCollection method), 1060
check_update() (matplotlib.collections.LineCollection method), 1072
check_update() (matplotlib.collections.PatchCollection method), 1083
check_update() (matplotlib.collections.PathCollection method), 1093
check_update() (matplotlib.collections.PolyCollection method), 1104
check_update() (matplotlib.collections.QuadMesh method), 1115
check_update() (matplotlib.collections.RegularPolyCollection method), 1125
check_update() (matplotlib.collections.StarPolygonCollection method), 1136
check_update() (matplotlib.collections.TriangleCollection method), 1147
check_update() (matplotlib.image.PcolorImage method), 1239
Char (class in matplotlib.mathtext), 1268
Matplotlib, Release 1.4.3

method), 1125
check_update() (matplotlib.collections.StarPolygonCollection
method), 1137
check_update() (matplotlib.collections.TriMesh
method), 1147
CheckButtons (class in matplotlib.widgets), 1616
checksum (matplotlib.dviread.Tfm attribute), 1189
Circle (class in matplotlib.patches), 1336
circle() (matplotlib.path.Path class method), 1366
circle_ratios() (matplotlib.tri.TriAnalyzer method),
1607
CircleCollection (class in matplotlib.collections),
1027
CirclePolygon (class in matplotlib.patches), 1337
circular_spine() (matplotlib.spines.Spine class
method), 1568
ccla() (in module matplotlib.pyplot), 1411
ccla() (matplotlib.axes.Axes method), 804
ccla() (matplotlib.axis.Axes method), 953
ccla() (matplotlib.spines.Spine method), 1568
ccla (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 557, 615
clabel() (in module matplotlib.pyplot), 1411
clabel() (matplotlib.axes.Axes method), 804
clabel() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 557, 615
clamp() (matplotlib.mathtext.Ship static method),
1278
clean() (matplotlib.cbook.Grouper method), 989
cleaned() (matplotlib.path.Path method), 1367
cleanup() (matplotlib.animation.FileMovieWriter
method), 749
cleanup() (matplotlib.animation.MovieWriter
method), 752
cleanup_path() (in module matplotlib.path), 1371
clear() (matplotlib.axes.Axes method), 811
clear() (matplotlib.cbook.MemoryMonitor method),
989
clear() (matplotlib.cbook.Stack method), 991
clear() (matplotlib.figure.Figure method), 1196
clear() (matplotlib.transforms.Affine2D method),
456
clear() (matplotlib.widgets.Cursor method), 1617
clear() (matplotlib.widgets.MultiCursor method),
1619
clf() (in module matplotlib.pyplot), 1418
clf() (matplotlib.figure.Figure method), 1196
clim() (in module matplotlib.pyplot), 1418
clip_path_to_rect() (in module matplotlib.path),
1371
clip_to_bbox() (matplotlib.path.Path method), 1367
close() (in module matplotlib.pyplot), 1418
close() (matplotlib.backends.backend_pdf.PdfPages
method), 984
close() (matplotlib.dviread.Dvi method), 1187
close_event() (matplotlib.backend_bases.FigureCanvasBase
method), 964
close_group() (matplotlib.backend_bases.FigureCanvasBase
method), 977
CloseEvent (class in matplotlib.backend_bases), 963
CLOSEPOLY (matplotlib.path.Path attribute), 1366
cm_fallback (matplotlib.mathtext.StixFonts attribute),
1279
cmap (matplotlib.cm.ScalarMappable attribute),
1001
code_type (matplotlib.path.Path attribute), 1367
codes (matplotlib.legend.Legend attribute), 1245
codes (matplotlib.path.Path attribute), 1367
cohere() (in module matplotlib.mlab), 1290
cohere() (in module matplotlib.pyplot), 1419
cohere() (matplotlib.axes.Axes method), 811
cohere_pairs() (in module matplotlib.mlab), 1291
Collection (class in matplotlib.collections), 1038
color() (in module matplotlib.collections.EventQueueCollection
method), 1061
color() (matplotlib.collections.LineCollection
method), 1072
Colorbar (class in matplotlib.colorbar), 1159
colorbar (matplotlib.cm.ScalarMappable attribute),
1001
colorbar() (in module matplotlib.pyplot), 1421
colorbar() (matplotlib.figure.Figure method), 1196
colorbar_extend (matplotlib.colors.Colormap
attribute), 1167
colorbar_factory() (in module matplotlib.colorbar),
1161
ColorbarBase (class in matplotlib.colorbar), 1160
ColorbarPatch (class in matplotlib.colorbar), 1161
ColorConverter (class in matplotlib.colors), 1166
Colormap (class in matplotlib.colors), 1167
colormaps() (in module matplotlib.pyplot), 696
colors (matplotlib.colors.ColorConverter attribute),
1166

2542 Index
colors() (in module matplotlib.pyplot), 1424
complex_spectrum() (in module matplotlib.mlab), 1292
composite_transform_factory() (in module matplotlib.transforms), 462
CompositeAffine2D (class in matplotlib.transforms), 462
CompositeGenericTransform (class in matplotlib.transforms), 461
config_axis() (matplotlib.colorbar.ColorbarBase method), 1160
connect() (in module matplotlib.pyplot), 1424
connect() (matplotlib.cbook.CallbackRegistry method), 988
connect() (matplotlib.patches.ConnectionStyle.Angle method), 1341
connect() (matplotlib.patches.ConnectionStyle.Angle3 method), 1341
connect() (matplotlib.patches.ConnectionStyle.Arc method), 1342
connect() (matplotlib.patches.ConnectionStyle.Arc3 method), 1342
connect() (matplotlib.patches.ConnectionStyle.Bar method), 1342
connect() (matplotlib.widgets.MultiCursor method), 1619
connect_event() (matplotlib.widgets.AxesWidget method), 1615
ConnectionPatch (class in matplotlib.patches), 1338
ConnectionStyle (class in matplotlib.patches), 1340
ConnectionStyle.Angle (class in matplotlib.patches), 1341
ConnectionStyle.Angle3 (class in matplotlib.patches), 1341
ConnectionStyle.Arc (class in matplotlib.patches), 1341
ConnectionStyle.Arc3 (class in matplotlib.patches), 1342
ConnectionStyle.Bar (class in matplotlib.patches), 1342
contains() (matplotlib.artist.Artist method), 757
contains() (matplotlib.axes.Axes method), 814
contains() (matplotlib.axis.Tick method), 958
contains() (matplotlib.axis.XAxis method), 959
contains() (matplotlib.axis.YAxis method), 960
contains() (matplotlib.collections.AsteriskPolygonCollection method), 1006
contains() (matplotlib.collections.CircleCollection method), 1017
contains() (matplotlib.collections.Collection method), 1028
contains() (matplotlib.collections.EllipseCollection method), 1039
contains() (matplotlib.collections.EventCollection method), 1049
contains() (matplotlib.collections.LineCollection method), 1061
contains() (matplotlib.collections.PatchCollection method), 1072
contains() (matplotlib.collections.PathCollection method), 1083
contains() (matplotlib.collections.QuadMesh method), 1093
contains() (matplotlib.collections.RegularPolyCollection method), 1104
contains() (matplotlib.collections.TriMesh method), 1115
contains() (matplotlib.collections.RegularPolyCollection method), 1115
contains() (matplotlib.collections.StarPolygonCollection method), 1126
contains() (matplotlib.collections.TriMesh method), 1137
contains() (matplotlib.collections.TriMesh method), 1147
contains() (matplotlib.figure.Figure method), 1199
contains() (matplotlib.image.BboxImage method), 1238
contains() (matplotlib.image.FigureImage method), 1238
contains() (matplotlib.legend.Legend method), 1245
contains() (matplotlib.lines.Line2D method), 1252
contains() (matplotlib.offsetbox.AnnotationBbox method), 1317
contains() (matplotlib.offsetbox.OffsetBox method), 1321
contains() (matplotlib.patches.Ellipse method), 1343
contains() (matplotlib.patches.Patch method), 1351
contains() (matplotlib.patches.Rectangle method), 1358
contains() (matplotlib.text.Annotation method), 1576
contains() (matplotlib.text.Text method), 1577
contains() (matplotlib.transforms.BboxBase method), 446
contains() (matplotlib.transforms.BboxBase method), 451
contains() (matplotlib.transforms.BboxBase method), 451
plotlib.transforms.Transform method), 451
contains_path() (matplotlib.path.Path method), 1367
contains_point() (matplotlib.axes.Axes method), 814
convert_units() (matplotlib.artist.Artist method), 757
convert_units() (matplotlib.transforms.Transform method), 451
contains_path() (matplotlib.path.Path method), 1367
contains_point() (matplotlib.patches.Patch method), 1351
contains_point() (matplotlib.path.Path method), 1367
contains_points() (matplotlib.path.Path method), 1368
containsx() (matplotlib.transforms.BboxBase method), 446
convert_yunits() (matplotlib.artist.Artist method), 757
convert_xunits() (matplotlib.transforms.BboxBase method), 446
context() (in module matplotlib.style), 1571
contiguous_regions() (in module matplotlib.mlab), 1293
contour() (in module matplotlib.pyplot), 1425
contour() (matplotlib.axes.Axes method), 814
convert_units() (matplotlib.transforms.BboxBase method), 446
contourf() (in module matplotlib.pyplot), 1436
contour() (mpl_toolkits.mplot3d.Axes3D method), 542, 602
convert_xunits() (mpl_toolkits.mplot3d.Axes3D method), 557, 615
contour3D() (mpl_toolkits.mplot3d.Axes3D method), 557, 616
convert_xunits() (mpl_toolkits.mplot3d.Axes3D method), 542, 602
contour() (mpl_toolkits.mplot3d.Axes3D method), 557, 615
contour3D() (mpl_toolkits.mplot3d.Axes3D method), 557, 616
convert_xunits() (mpl_toolkits.mplot3d.Axes3D method), 557, 615
contourf() (mpl_toolkits.mplot3d.Axes3D method), 545, 605
convert_xunits() (mpl_toolkits.mplot3d.Axes3D method), 557, 615
contourf3D() (mpl_toolkits.mplot3d.Axes3D method), 558, 616
ConversionInterface (class in matplotlib.units), 1614
convert() (matplotlib.units.ConversionInterface static method), 1614
convertMeshToPaths() (matplotlib.collections.QuadMesh static method), 1115
convertMeshToTriangles() (matplotlib.collections.QuadMesh static method), 1115
ConversionInterface (class in matplotlib.units), 1614
convert() (matplotlib.units.ConversionInterface static method), 1614
convertMeshToPaths() (matplotlib.collections.TriMesh static method), 1147
convertMeshToTriangles() (matplotlib.collections.TriMesh static method), 1147
convert_path_to_polygons() (in module matplotlib.path), 1371
convert_units() (matplotlib.axis.Axis method), 953
convert_xunits() (matplotlib.transforms.Transform method), 757
convert_xunits() (matplotlib.collections.AsteriskPolygonCollection method), 1007
convert_xunits() (matplotlib.collections.BrokenBarHCollection method), 1017
convert_xunits() (matplotlib.collections.CircleCollection method), 1028
convert_xunits() (matplotlib.collections.Collection method), 1039
convert_xunits() (matplotlib.collections.EllipseCollection method), 1049
convert_xunits() (matplotlib.collections.EventCollection method), 1061
convert_xunits() (matplotlib.collections.LineCollection method), 1073
convert_xunits() (matplotlib.collections.PathCollection method), 1093
convert_xunits() (matplotlib.collections.PatchCollection method), 1104
convert_xunits() (matplotlib.collections.QuadMesh method), 1115
convert_xunits() (matplotlib.collections.RegularPolyCollection method), 1126
convert_xunits() (matplotlib.collections.StarPolygonCollection method), 1137
convert_xunits() (matplotlib.collections.TriMesh method), 1147
convert_xunits() (matplotlib.artist.Artist method), 757
convert_xunits() (matplotlib.axes.Axes method), 836
convert_xunits() (matplotlib.collections.AsteriskPolygonCollection method), 1007
...method), 1589
createFontList() (in module matplotlib.font_manager), 1230
cross_from_above() (in module matplotlib.mlab), 1293
cross_from_below() (in module matplotlib.mlab), 1293
csd() (in module matplotlib.mlab), 1294
csd() (in module matplotlib.pyplot), 1447
csd() (matplotlib.axes.Axes method), 836
csv2rec() (in module matplotlib.mlab), 1295
csvformat_factory() (in module matplotlib.mlab), 1296
CubicTriInterpolator (class in matplotlib.tri), 1602
current_key_axes() (matplotlib.figure.AxesStack method), 1191
cursor (class in matplotlib.widgets), 1617
cursors (class in matplotlib.backend_bases), 963
CURVE3 (matplotlib.path.Path attribute), 1366
CURVE4 (matplotlib.path.Path attribute), 1366
customspace() (matplotlib.mathtext.Parser method), 1276

D
dashd (matplotlib.backend_bases.GraphicsContextBase attribute), 970
datalim_to_dt() (matplotlib.dates.DateLocator method), 1179
date2num() (in module matplotlib.dates), 1177
DateFormatter (class in matplotlib.dates), 1177
DateLocator (class in matplotlib.dates), 1179
dateutil, 2529
DayLocator (class in matplotlib.dates), 1181
dblclick (matplotlib.backend_bases.MouseEvent attribute), 974
dedent() (in module matplotlib.cbook), 993
deprecated() (in module matplotlib.cbook), 994
detrend() (in module matplotlib.mlab), 1297
detrend_linear() (in module matplotlib.mlab), 1297
detrend_mean() (in module matplotlib.mlab), 1297
detrend_none() (in module matplotlib.mlab), 1298
dict_delall() (in module matplotlib.cbook), 994
disable_mouse_rotation() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 558, 617
disconnect() (in module matplotlib.pyplot), 1450
disconnect() (matplotlib.cbook.CallbackRegistry method), 988
disconnect() (matplotlib.offsetbox.DraggableBase method), 1319
disconnect() (matplotlib.widgets.Button method), 1616
disconnect() (matplotlib.widgets.CheckButtons method), 1616
disconnect() (matplotlib.widgets.MultiCursor method), 1620
disconnect() (matplotlib.widgets.RadioButtons method), 1620
disconnect() (matplotlib.widgets.Slider method), 1623
disconnect_events() (matplotlib.widgets.AxesWidget method), 1615
dist() (in module matplotlib.mlab), 1298
dist_point_to_segment() (in module matplotlib.mlab), 1298
distances_along_curve() (in module matplotlib.mlab), 1298
Divider (class in mpl_toolkits.axes_grid.axes_divider), 520, 680
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Line3DCollection method), 578, 635
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Patch3D...
Dvi (class in matplotlib.dviread), 1187
DviFont (class in matplotlib.dviread), 1187
dynamic_update() (matplotlib.backend_bases.NavigationToolbar2 method), 975

E
edges (matplotlib.tri.Triangulation attribute), 1599
Ellipse (class in matplotlib.patches), 1342
EllipseCollection (class in matplotlib.collections), 1048
embedTTF() (matplotlib.backends.backend_pdf.PdfFile method), 983
empty() (matplotlib.cbook.Stack method), 991
Encoding (class in matplotlib.dviread), 1188
encoding (matplotlib.dviread.Encoding attribute), 1188
end() (matplotlib.backends.backend_pdf.Stream method), 985
end_group() (matplotlib.mattext.Parser method), 1276
end_pan() (matplotlib.axes.Axes method), 840
enter_notify_event() (matplotlib.backend_bases.FigureCanvasBase method), 965
entropy() (in module matplotlib.mlab), 1298
environment variable
HOME, 380, 383
MPLCONFIGDIR, 380, 383
PATH, 52, 55, 56, 60, 383
PYTHONPATH, 383, 399
epoch2num() (in module matplotlib.dates), 1177
EPS, 2529
Error() (in module matplotlib.mattext), 1268
errorbar() (in module matplotlib.pyplot), 1451
errorbar() (matplotlib.axes.Axes method), 840
evaluate() (matplotlib.mlab.GaussianKDE method), 1288
Event (class in matplotlib.backend_bases), 964
EventCollection (class in matplotlib.collections), 1059
eventplot() (in module matplotlib.pyplot), 1454
eventplot() (matplotlib.axes.Axes method), 843
events (matplotlib.backend_bases.FigureCanvasBase attribute), 965
eventson (matplotlib.widgets.Widget attribute), 1625
exception_to_str() (in module matplotlib.cbook), 994
exec_key (matplotlib.animation.AVConvBase attribute), 747
exec_key (matplotlib.animation.FFMpegBase attribute), 749
exec_key (matplotlib.animation.ImageMagickBase attribute), 750
exec_key (matplotlib.animation.MencoderBase attribute), 751
exp_safe() (in module matplotlib.mlab), 1299
expanded() (matplotlib.transforms.BboxBase method), 446
extend_positions() (matplotlib.collections.EventCollection method), 1061
extents (matplotlib.transforms.BboxBase attribute), 446

F
factory() (matplotlib.mattext.GlueSpec class method), 1271
family_escape() (in module matplotlib.fontconfig_pattern), 1231
family_unescape() (in module matplotlib.fontconfig_pattern), 1231
FancyArrow (class in matplotlib.patches), 1343
FancyArrowPatch (class in matplotlib.patches), 1344
FancyBboxPatch (class in matplotlib.patches), 1347
fetch_historical_yahoo() (in module matplotlib.finance), 1216
FFMpegBase (class in matplotlib.animation), 748
FFMpegFileWriter (class in matplotlib.animation), 749
FFMpegWriter (class in matplotlib.animation), 749
fftsurr() (in module matplotlib.mlab), 1299
FIFOBuffer (class in matplotlib.mlab), 1285
figaspect() (in module matplotlib.figure), 1211
figimage() (in module matplotlib.pyplot), 1456
figimage() (matplotlib.figure.Figure method), 1199
figlegend() (in module matplotlib.pyplot), 1458
figtext() (in module matplotlib.pyplot), 1458
Figure (class in matplotlib.figure), 1191
figure() (in module matplotlib.pyplot), 1460
FigureCanvas (in module matplotlib.backends.backend_pdf), 983
FigureCanvas (in module matplotlib.backends.backend_qt4agg), 981
752 finish() (matplotlib.sankey.Sankey method), 1564
fix_minus() (matplotlib.ticker.Formatter method), 1589
fix_minus() (matplotlib.ticker.ScalarFormatter method), 1590
Fixed (class in mpl_toolkits.axes_grid.axes_size), 519, 679
fixed_dpi (matplotlib.backend_bases.FigureCanvasBase attribute), 965
FixedFormatter (class in matplotlib.ticker), 1590
FixedLocator (class in matplotlib.ticker), 1593
flag() (in module matplotlib.pyplot), 1469
flatten() (in module matplotlib.cbook), 995
flipy() (matplotlib.backend_bases.RendererBase method), 978
flush_events() (matplotlib.backend_bases.FigureCanvasBase method), 965
font (matplotlib.mathtext.Parser.State attribute), 1276
font() (matplotlib.mathtext.Parser method), 1276
FontconfigPatternParser (class in matplotlib.fontconfig_pattern), 1231
FontEntry (class in matplotlib.font_manager), 1225
FontManager (class in matplotlib.font_manager), 1225
fontmap (matplotlib.mathtext.StandardPsFonts attribute), 1278
fontName() (matplotlib.backends.backend_pdf.PdfFile frame_format (matplotlib.animation.FileMovieWriter attribute), 749
format_data_short() (matplotlib.ticker.LogFormatter method), 1591
format_data_short() (matplotlib.ticker.ScalarFormatter method), 1590
format_xdata() (matplotlib.axes.Axes method), 854
format_ydata() (matplotlib.axes.Axes method), 854
format_zdata() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 558, 617
FormattingBool (class in matplotlib.mlab), 1286
FormatDate (class in matplotlib.mlab), 1286
FormatDatet ime (class in matplotlib.mlab), 1286
FormatF loat (class in matplotlib.mlab), 1286
FormatF ormatStr (class in matplotlib.mlab), 1286
FormatInt (class in matplotlib.mlab), 1286
FormatMillions (class in matplotlib.mlab), 1286
FormatObj (class in matplotlib.mlab), 1286
FormatPercent (class in matplotlib.mlab), 1287
FormatStrFormatter (class in matplotlib.ticker), 1590
FormatString (class in matplotlib.mlab), 1287
formatter (class in matplotlib.ticker), 1589
formatter (class in matplotlib.ticker), 1589
formatter (class in matplotlib.axis.Ticker attribute), 959
FormatThousands (class in matplotlib.mlab), 1287
forward() (matplotlib.backend_bases.NavigationToolbar2 method), 975
forward() (matplotlib.cbook.Stack method), 991
frac() (matplotlib.mathtext.Parser method), 1276
Fraction (class in mpl_toolkits.axes_grid.axes_size), 519, 679
frame_format (matplotlib.animation.MovieWriter attribute), 752
frame_size (matplotlib.animation.MovieWriter attribute), 752
frange() (in module matplotlib.mlab), 1299
freetype (2529) in module matplotlib.mlab), 1299
from_any() (in module mpl_toolkits.mplot3d.axes3d.Axes3D method), 472
from_bounds() (matplotlib.transforms.Bbox static method), 449
from_extents() (matplotlib.transforms.Bbox static method), 449
from_images() (in module matplotlib.image), 1240
from_levels_and_colors() (in module matplotlib.colors), 1172
from_list() (matplotlib.colors.LinearSegmentedColormap static method), 1169

Index 2551
from_values() (matplotlib.transforms.AffineTransform static method), 456
fromarray() (in module matplotlib.image), 1240
fromarray2() (in module matplotlib.image), 1240
frombuffer() (in module matplotlib.image), 1240
frombyte() (in module matplotlib.image), 1240
fromstr() (matplotlib.mlab.FormatBool method), 1286
fromstr() (matplotlib.mlab.FormatDate method), 1286
fromstr() (matplotlib.mlab.FormatDatetime method), 1286
fromstr() (matplotlib.mlab.FormatFloat method), 1286
fromstr() (matplotlib.mlab.FormatInt method), 1286
fromstr() (matplotlib.mlab.FormatObj method), 1287
frozen() (matplotlib.transforms.AffineTransform method), 455
frozen() (matplotlib.transforms.BboxBase method), 446
frozen() (matplotlib.transforms.BlendedGenericTransform method), 460
frozen() (matplotlib.transforms.CompositeGenericTransform method), 461
frozen() (matplotlib.transforms.IdentityTransform method), 458
frozen() (matplotlib.transforms.TransformNode method), 445
frozen() (matplotlib.transforms.TransformWrapper method), 454
full_screen_toggle() (matplotlib.backends.backend_bases.FigureManagerBase method), 969
fully_contains() (matplotlib.transforms.BboxBase method), 446
fully_containsx() (matplotlib.transforms.BboxBase method), 446
fully_containsy() (matplotlib.transforms.BboxBase method), 446
fully_overlaps() (matplotlib.transforms.BboxBase method), 446
funcAnimation (class in matplotlib.animation), 750
funcbottom() (matplotlib.widgets.SubplotTool method), 1624
funcleft() (matplotlib.widgets.SubplotTool method), 1624
funcright() (matplotlib.widgets.SubplotTool method), 1625
function() (matplotlib.mathtext.Parser method), 1276
funcbottom() (matplotlib.widgets.SubplotTool method), 1625
funcwspace() (matplotlib.widgets.SubplotTool method), 1625

G
GaussianKDE (class in matplotlib.mlab), 1287
gca() (in module matplotlib.pyplot), 1469
get() (matplotlib.figure.Figure method), 1201
gcf() (in module matplotlib.pyplot), 1470
gci() (in module matplotlib.pyplot), 1470
GDK, 2529
generate_fontconfig_pattern() (in module matplotlib.fontconfig_pattern), 1231
genfrac() (matplotlib.mathtext.Parser method), 1276
get() (in module matplotlib.pyplot), 1470
get() (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_adjustable() (matplotlib.axes.Axes method), 854
get() (matplotlib.transforms.AffineTransform method), 454
get() (matplotlib.transforms.BlendedGenericTransform method), 460
get() (matplotlib.transforms.CompositeGenericTransform method), 461
get() (matplotlib.transforms.IdentityTransform method), 458
get() (matplotlib.transforms.Transform method), 451
get_agg_filter() (matplotlib.artist.Artist method), 757
get_agg_filter() (matplotlib.axes.Axes method), 855
get_agg_filter() (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_agg_filter() (matplotlib.collections.BrokenBarHCollection method), 1018
get_agg_filter() (matplotlib.collections.CircleCollection method), 1028
get_agg_filter() (matplotlib.collections.EllipseCollection method), 1050
get_agg_filter() (matplotlib.collections.EventCollection method), 1061
get_agg_filter() (matplotlib.collections.LineCollection method), 1073
get_agg_filter() (matplotlib.collections.PatchCollection method), 1083
get_agg_filter() (matplotlib.collections.PathCollection method), 1094
get_agg_filter() (matplotlib.collections.PolyCollection method), 1104
get_agg_filter() (matplotlib.collections.QuadMesh method), 1115
get_agg_filter() (matplotlib.collections.RegularPolyCollection method), 1126
get_agg_filter() (matplotlib.collections.StarPolygonCollection method), 1137
get_agg_filter() (matplotlib.collections.TriMesh method), 1148
get_alpha() (matplotlib.collections.Collection method), 1039
get_alpha() (matplotlib.collections.EllipseCollection method), 1050
get_alpha() (matplotlib.collections.EventCollection method), 1061
get_alpha() (matplotlib.collections.LineCollection method), 1073
get_alpha() (matplotlib.collections.PatchCollection method), 1083
get_alpha() (matplotlib.collections.PathCollection method), 1094
get_alpha() (matplotlib.collections.PolyCollection method), 1104
get_alpha() (matplotlib.collections.QuadMesh method), 1115
get_alpha() (matplotlib.collections.RegularPolyCollection method), 1126
get_alpha() (matplotlib.collections.StarPolygonCollection method), 1137
get_alpha() (matplotlib.collections.TriMesh method), 1148
get_anchored() (matplotlib.artist.Artist method), 757
get_anchored() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_angle() (matplotlib.afm.AFM method), 743
get_animated() (matplotlib.artist.Artist method), 757
get_animated() (matplotlib.axes.Axes method), 855
get_animated() (matplotlib.backend_bases.GraphicsContextBase method), 970
get_animated() (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_animated() (matplotlib.collections.BrokenBarHCollection method), 1018
get_animated() (matplotlib.collections.CircleCollection method), 1028
get_animated() (matplotlib.collections.EllipseCollection method), 1050
get_animated() (matplotlib.collections.LineCollection method), 1073
get_animated() (matplotlib.collections.PatchCollection method), 1083
get_animated() (matplotlib.collections.PathCollection method), 1094
get_animated() (matplotlib.collections.PolyCollection method), 1104
get_animated() (matplotlib.collections.QuadMesh method), 1115
get_animated() (matplotlib.collections.RegularPolyCollection method), 1126
get_animated() (matplotlib.collections.StarPolygonCollection method), 1137
get_animated() (matplotlib.collections.TriMesh method), 1148
get_anchored() (matplotlib.axes.Axes method), 855
get_anchored() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_angle() (matplotlib.afm.AFM method), 743
get_animated() (matplotlib.artist.Artist method), 757
get_animated() (matplotlib.axes.Axes method), 855
get_animated() (matplotlib.backend_bases.GraphicsContextBase method), 970
get_animated() (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_animated() (matplotlib.collections.BrokenBarHCollection method), 1018
get_animated() (matplotlib.collections.CircleCollection method), 1028
get_animated() (matplotlib.collections.EllipseCollection method), 1050
get_animated() (matpl-
plotlib.collections.EventCollection method), 1061
get_animated() (matplotlib.collections.LineCollection method), 1073
get_animated() (matplotlib.collections.PatchCollection method), 1084
get_animated() (matplotlib.collections.PathCollection method), 1094
get_animated() (matplotlib.collections.PolyCollection method), 1105
get_animated() (matplotlib.collections.QuadMesh method), 1115
get_animated() (matplotlib.collections.RegularPolyCollection method), 1126
get_animated() (matplotlib.collections.StarPolygonCollection method), 1137
get_animated() (matplotlib.collections.TriMesh method), 1148
get_array() (matplotlib.collections.LineCollection method), 1073
get_array() (matplotlib.collections.PatchCollection method), 1084
get_array() (matplotlib.collections.PathCollection method), 1094
get_array() (matplotlib.collections.PolyCollection method), 1105
get_array() (matplotlib.collections.QuadMesh method), 1115
get_array() (matplotlib.collections.RegularPolyCollection method), 1126
get_array() (matplotlib.collections.StarPolygonCollection method), 1137
get_array() (matplotlib.collections.TriMesh method), 1148
get_array() (matplotlib.collections.EventCollection method), 1061
get_array() (matplotlib.collections.LineCollection method), 1073
get_array() (matplotlib.collections.PatchCollection method), 1084
get_array() (matplotlib.collections.PathCollection method), 1094
get_array() (matplotlib.collections.PolyCollection method), 1105
get_array() (matplotlib.collections.QuadMesh method), 1115
get_array() (matplotlib.collections.RegularPolyCollection method), 1126
get_array() (matplotlib.collections.StarPolygonCollection method), 1137
get_array() (matplotlib.collections.TriMesh method), 1148
get_arrowstyle() (matplotlib.patches.FancyArrowPatch method), 1346
get_aspect() (matplotlib.axes.Axes method), 855
get_aspect() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_autoscale_on() (matplotlib.axes.Axes method), 855
get_autoscale_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 558, 617
get_autoscalex_on() (matplotlib.axes.Axes method), 855
get_autoscaley_on() (matplotlib.axes.Axes method), 855
get_autoscalez_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 558, 617
get_axes() (matplotlib.artist.Artist method), 757
get_axes() (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_axes() (matplotlib.collections.BrokenBarHCollection method), 1018
get_axes() (matplotlib.collections.CircleCollection method), 1029
get_axes() (matplotlib.collections.Collection method), 1040
get_axes() (matplotlib.collections.EllipseCollection method), 1050
get_axes() (matplotlib.collections.EventCollection method), 1061
get_axes() (matplotlib.collections.LineCollection method), 1061
Index 2555
get_clip_box()  (matplotlib.collections.TriMesh method), 1148
get_clip_on()  (matplotlib.artist.Artist method), 758
get_clip_on()  (matplotlib.axes.Axes method), 855
get_clip_on()  (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_clip_on()  (matplotlib.collections.BrokenBarHCollection method), 1018
get_clip_on()  (matplotlib.collections.CircleCollection method), 1029
get_clip_on()  (matplotlib.collections.EllipseCollection method), 1050
get_clip_on()  (matplotlib.collections.EventCollection method), 1062
get_clip_on()  (matplotlib.collections.LineCollection method), 1073
get_clip_on()  (matplotlib.collections.PatchCollection method), 1084
get_clip_on()  (matplotlib.collections.PathCollection method), 1094
get_clip_on()  (matplotlib.collections.PolyCollection method), 1105
get_clip_on()  (matplotlib.collections.QuadMesh method), 1116
get_clip_on()  (matplotlib.collections.RegularPolyCollection method), 1126
get_clip_on()  (matplotlib.collections.StarPolygonCollection method), 1137
get_clip_on()  (matplotlib.collections.TriMesh method), 1148
get_clip_path()  (matplotlib.collections.BrokenBarHCollection method), 1018
get_clip_path()  (matplotlib.collections.CircleCollection method), 1029
get_clip_path()  (matplotlib.collections.Collection method), 1040
get_clip_path()  (matplotlib.collections.EllipseCollection method), 1050
get_clip_path()  (matplotlib.collections.EventCollection method), 1062
get_clip_path()  (matplotlib.collections.LineCollection method), 1073
get_clip_path()  (matplotlib.collections.PatchCollection method), 1084
get_clip_path()  (matplotlib.collections.PathCollection method), 1094
get_clip_path()  (matplotlib.collections.PolyCollection method), 1105
get_clip_path()  (matplotlib.collections.QuadMesh method), 1116
get_clip_path()  (matplotlib.collections.RegularPolyCollection method), 1126
get_clip_path()  (matplotlib.collections.StarPolygonCollection method), 1137
get_clip_path()  (matplotlib.collections.TriMesh method), 1148
get_clip_rectangle()  (matplotlib.backend_bases.GraphicsContextBase method), 970
get_cmap()  (in module matplotlib.cm), 1002
get_cmap()  (matplotlib.cm.ScalarMappable method), 1002
get_cmap()  (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_cmap()  (matplotlib.collections.BrokenBarHCollection method), 1018
get_closed()  (matplotlib.patches.Polygon method), 1356
get_closed()  (matplotlib.patches.Polygon method), 1356
get_cmap()  (in module matplotlib.cm), 1002
get_cmap()  (matplotlib.cm.ScalarMappable method), 1002
get_cmap()  (matplotlib.collections.AsteriskPolygonCollection method), 1007
get_cmap()  (matplotlib.collections.BrokenBarHCollection method), 1018
plotlib.collections.StarPolygonCollection
method), 1138

get_datalim() (matplotlib.collections.TriMesh
method), 1148

get_default_bbox_extra_artists()
(matplotlib.collections.Axes method), 856

get_default_bbox_extra_artists()
(matplotlib.figure.Figure method), 1202

get_default_filename()
(matplotlib.backend_bases.FigureCanvasBase
method), 965

get_default_filetype()
(matplotlib.backend_bases.FigureCanvasBase
class method), 965

get_default_handler_map()
(matplotlib.legend.Legend class method),
1245

get_default_size()
(matplotlib.font_manager.FontManager static
method), 1226

get_default_weight()
(matplotlib.font_manager.FontManager
method), 1226

get_depth() (matplotlib.mathtext.MathTextParser
method), 1272

get_dir_vector() (in module
mpl_toolkits.mplot3d.art3d), 581, 638

gdapixel() (matplotlib.figure.Figure method), 1202

gdapi_cor() (matplotlib.patches.FancyArrowPatch
method), 1346

gdawdrawstiles() (matplotlib.lines.Line2D
method), 1253

gdawec() (matplotlib.patches.Patch method), 1352

gdawedgecolor() (matplotlib.collections.AsteriskPolygonCollection
method), 1008

gdawedgecolor() (matplotlib.collections.BrokenBarHCollection
method), 1018

gdawedgecolor() (matplotlib.collections.CircleCollection
method), 1029

gdawedgecolor() (matplotlib.collections.Collection
method), 1040

gdawedgecolor() (matplotlib.collections.EllipseCollection
method), 1051

gdawedgecolor() (matplotlib.collections.EventCollection
method), 1062

gdawedgecolor() (matplotlib.collections.LineCollection
method), 1074

gdawedgecolor() (matplotlib.collections.PatchCollection
method), 1084

gdawedgecolor() (matplotlib.collections.PathCollection
method), 1094

gdawedgecolor() (matplotlib.collections.PolyCollection
method), 1105

gdawedgecolor() (matplotlib.collections.QuadMesh
method), 1116

gdawedgecolor() (matplotlib.collections.RegularPolyCollection
method), 1127

gdawedgecolor() (matplotlib.collections.StarPolygonCollection
method), 1138

gdawedgecolor() (matplotlib.collections.TriMesh
method), 1148

gdawedgecolor() (matplotlib.figure.Figure method), 1203

gdawedgecolor() (matplotlib.patches.Patch method), 1352

gdawedgecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection
method), 580, 637

gdawedgecolors() (matplotlib.collections.AsteriskPolygonCollection
method), 1008

gdawedgecolors() (matplotlib.collections.BrokenBarHCollection
method), 1018

gdawedgecolors() (matplotlib.collections.CircleCollection
method), 1029

gdawedgecolors() (matplotlib.collections.Collection
method), 1040

gdawedgecolors() (matplotlib.collections.EllipseCollection
method), 1051

gdawedgecolors() (matplotlib.collections.EventCollection
method), 1062

gdawedgecolors() (matplotlib.collections.PolyCollection
method), 1105

gdawedgecolors() (matplotlib.collections.QuadMesh
method), 1116

gdawedgecolors() (matplotlib.collections.RegularPolyCollection
method), 1127

gdawedgecolors() (matplotlib.collections.StarPolygonCollection
method), 1138

gdawedgecolors() (matplotlib.collections.TriMesh
method), 1148

gdawedgecolors() (matplotlib.figure.Figure method), 1203

gdawedgecolors() (matplotlib.patches.Patch method), 1352

gdawedgecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection
method), 580, 637

gdawedgecolors() (matplotlib.collections.AsteriskPolygonCollection
method), 1008

gdawedgecolors() (matplotlib.collections.BrokenBarHCollection
method), 1018

gdawedgecolors() (matplotlib.collections.CircleCollection
method), 1029

gdawedgecolors() (matplotlib.collections.Collection
method), 1040

gdawedgecolors() (matplotlib.collections.EllipseCollection
method), 1051

gdawedgecolors() (matplotlib.collections.EventCollection
method), 1062

2560 Index
get_ha() (matplotlib.text.Text method), 1578
get_hatch() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_hatch() (matplotlib.collections.BrokenBarHCollection method), 1019
get_hatch() (matplotlib.collections.CircleCollection method), 1029
get_hatch() (matplotlib.collections.Collection method), 1040
get_hatch() (matplotlib.collections.EllipseCollection method), 1051
get_hatch() (matplotlib.collections.EventCollection method), 1062
get_hatch() (matplotlib.collections.LineCollection method), 1074
get_hatch() (matplotlib.collections.PatchCollection method), 1084
get_hatch() (matplotlib.collections.PathCollection method), 1095
get_hatch() (matplotlib.collections.PolyCollection method), 1105
get_hatch() (matplotlib.collections.QuadMesh method), 1116
get_hatch() (matplotlib.collections.RegularPolyCollection method), 1127
get_hatch() (matplotlib.collections.TriMesh method), 1149
get_hatch() (matplotlib.patches.Patch method), 1352
get_hatch_path() (matplotlib.backend_bases.GraphicsContextBase method), 970
get_height() (matplotlib.patches.FancyBboxPatch method), 1349
get_height() (matplotlib.patches.Rectangle method), 1358
get_height_char() (matplotlib.afm.AFM method), 744
get_height_ratios() (matplotlib.gridspec.GridSpecBase method), 1234
get_helper() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 525, 685
get_hinting_type() (matplotlib.mathtext.MathTextBackend method), 1273
get_horizontal() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
generate() (mpl_toolkits.axes_grid.automatically_zoomed_axes.AxesDivider method), 753
get_horizontal_sizes() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_horizontal_stem_width() (matplotlib.afm.AFM method), 744
get_horizontalalignment() (matplotlib.text.Text method), 1578
get_image_magnification() (matplotlib.backend_bases.RendererBase method), 978
get_images() (matplotlib.axes.Axes method), 856
get_joinstyle() (matplotlib.backend_bases.GraphicsContextBase method), 970
get_joinstyle() (matplotlib.markers.MarkerStyle method), 1263
get_joinstyle() (matplotlib.patches.Patch method), 1352
generate() (mpl_toolkits.axes_grid.automatically_zoomed_axes.AxesDivider method), 753
get_kern() (matplotlib.mathtext.Fonts method), 1269
get_kern() (matplotlib.mathtext.StandardPsFonts method), 1278
generate() (mpl_toolkits.axes_grid.automatically_zoomed_axes.AxesDivider method), 753
get_kern_dist() (matplotlib.afm.AFM method), 744
get_kern_dist_from_name() (matplotlib.afm.AFM method), 744
get_kerning() (matplotlib.mathtext.Char method), 1268
get_kerning() (matplotlib.mathtext.Node method), 1275
generate() (mpl_toolkits.axes_grid.automatically_zoomed_axes.AxesDivider method), 753
get_label() (matplotlib.artist.Artist method), 758
get_label() (matplotlib.axes.Axes method), 856
get_label() (matplotlib.axis.Axis method), 953
get_label() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_label() (matplotlib.collections.BrokenBarHCollection method), 1019
get_label() (matplotlib.collections.CircleCollection method), 1029
get_label() (matplotlib.collections.Collection method), 1040
get_label() (matplotlib.collections.EllipseCollection method), 1051
get_label() (matplotlib.collections.EventCollection method), 1051
get_label() (matplotlib.collections.LineCollection method), 1074
get_label() (matplotlib.collections.PatchCollection method), 1085
get_label() (matplotlib.collections.PathCollection method), 1095
get_label() (matplotlib.collections.PolyCollection method), 1106
get_label() (matplotlib.collections.QuadMesh method), 1116
get_label() (matplotlib.collections.RegularPolyCollection method), 1127
get_label() (matplotlib.collections.StarPolygonCollection method), 1138
get_label() (matplotlib.collections.TriMesh method), 1149
get_label_position() (matplotlib.axis.XAxis method), 959
get_label_position() (matplotlib.axis.YAxis method), 960
get_label_text() (matplotlib.axis.Axis method), 953
get_legend() (matplotlib.axes.Axes method), 856
get_legend_handler() (matplotlib.legend.Legend static method), 1245
get_legend_handler_map() (matplotlib.legend.Legend method), 1245
get Legend_handler_labels() (matplotlib.axes.Axes method), 856
get_length() (matplotlib.collections.EventCollection method), 1062
get_lineoffset() (matplotlib.collections.EventCollection method), 1062
get_lines() (matplotlib.axes.Axes method), 856
get_lines() (matplotlib.legend.Legend method), 1246
get_linestyle() (matplotlib.backend_bases.GraphicsContextBase method), 970
get_linestyle() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_linestyle() (matplotlib.collections.BrokenBarHCollection method), 1019
get_linestyle() (matplotlib.collections.CircleCollection method), 1030
get_linestyle() (matplotlib.collections.Collection method), 1041
get_linestyle() (matplotlib.collections.EllipseCollection method), 1051
get_linestyle() (matplotlib.collections.EventCollection method), 1063
get_linestyle() (matplotlib.collections.LineCollection method), 1074
get_linestyle() (matplotlib.collections.PatchCollection method), 1085
get_linestyle() (matplotlib.collections.PathCollection method), 1095
get_linestyle() (matplotlib.collections.PolyCollection method), 1106
get_linestyle() (matplotlib.collections.QuadMesh method), 1116
get_linestyle() (matplotlib.collections.RegularPolyCollection method), 1127
get_linestyle() (matplotlib.collections.StarPolygonCollection method), 1138
get_linestyle() (matplotlib.collections.TriMesh method), 1149
get_linestyle() (matplotlib.lines.Line2D method), 1253
get_linestyle() (matplotlib.patches.Patch method), 1352
get_linestyles() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_linestyles() (matplotlib.collections.BrokenBarHCollection method), 1019
get_linestyles() (matplotlib.collections.CircleCollection method), 1030
get_linestyles() (matplotlib.collections.Collection method), 1041
get_linestyles() (matplotlib.collections.EllipseCollection method), 1051
get_linestyles() (matplotlib.collections.EventCollection method), 1063
get_linestyles() (matplotlib.collections.LineCollection method), 1074
get_linestyles() (matplotlib.collections.PatchCollection method), 1085
get_linestyles() (matplotlib.collections.PathCollection method), 1095
get_linestyles() (matplotlib.collections.PolyCollection method), 1106
get_linestyles() (matplotlib.collections.QuadMesh method), 1117
get_linestyles() (matplotlib.collections.RegularPolyCollection method), 1127
get_linestyles() (matplotlib.collections.StarPolygonCollection method), 1138
get_linestyles() (matplotlib.collections.TriMesh method), 1149
get_linewidth() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_linewidth() (matplotlib.collections.BrokenBarHCollection method), 1019
get_linewidth() (matplotlib.collections.CircleCollection method), 1030
get_linewidth() (matplotlib.collections.EllipseCollection method), 1051
get_linewidth() (matplotlib.collections.EventCollection method), 1063
get_linewidth() (matplotlib.collections.LineCollection method), 1074
get_linewidth() (matplotlib.collections.PatchCollection method), 1106
get_linewidth() (matplotlib.collections.PathCollection method), 1117
get_linewidth() (matplotlib.collections.PolyCollection method), 1127
get_linewidth() (matplotlib.collections.QuadMesh method), 1138
get_linewidth() (matplotlib.collections.RegularPolyCollection method), 1149
get_linewidth() (matplotlib.collections.StarPolygonCollection method), 1152
get_linewidth() (matplotlib.collections.TriMesh method), 1153
get_linewidth() (matplotlib.patches.Patch method), 1352
get_linewidths() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_linewidths() (matplotlib.collections.BrokenBarHCollection method), 1019
get_linewidths() (matplotlib.collections.CircleCollection method), 1030
get_linewidths() (matplotlib.collections.EllipseCollection method), 1051
get_linewidths() (matplotlib.collections.EventCollection method), 1063
get_linewidths() (matplotlib.collections.LineCollection method), 1074
get_linewidths() (matplotlib.collections.PatchCollection method), 1106
get_linewidths() (matplotlib.collections.PathCollection method), 1117
get_linewidths() (matplotlib.collections.PolyCollection method), 1127
get_linewidths() (matplotlib.collections.QuadMesh method), 1138
get_linewidths() (matplotlib.collections.RegularPolyCollection method), 1149
get_linewidths() (matplotlib.collections.StarPolygonCollection method), 1152
get_linewidths() (matplotlib.collections.TriMesh method), 1153
get_linewidths() (matplotlib.patches.Patch method), 1352
get_linewidths() (matplotlib.collections.PathCollection method), 1085
get_linewidths() (matplotlib.collections.PolyCollection method), 1106
get_linewidths() (matplotlib.collections.QuadMesh method), 1117
get_linewidths() (matplotlib.collections.RegularPolyCollection method), 1127
get_linewidths() (matplotlib.collections.StarPolygonCollection method), 1138
get_linewidths() (matplotlib.collections.TriMesh method), 1149
get_loc() (matplotlib.axis.Tick method), 958
get_loc_in_canvas() (matplotlib.offsetbox.DraggableOffsetBox method), 1319
get_locator() (matplotlib.axis.Axis method), 954
get_locator() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 576, 634
get_major_formatter() (matplotlib.axis.Axis method), 954
get_major_locator() (matplotlib.axis.Axis method), 954
get_major_ticks() (matplotlib.axis.Axis method), 954
get_major_ticks() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 576, 634
get_major_ticklabels() (matplotlib.axis.Axis method), 954
get_major_ticklines() (matplotlib.axis.Axis method), 954
get_major_ticklocs() (matplotlib.axis.Axis method), 954
get_marker() (matplotlib.lines.Line2D method), 1253
get_marker() (matplotlib.markers.MarkerStyle method), 1263
get_markeredgecolor() (matplotlib.lines.Line2D method), 1253
get_markerfacecolor() (matplotlib.lines.Line2D method), 1254
get_markerfacecoloralt() (matplotlib.lines.Line2D method), 1254
get_markersize() (matplotlib.lines.Line2D method), 1254
get_markerevery() (matplotlib.lines.Line2D method), 1254
get_masked_triangles() (matplotlib.tri.Triangulation method), 1600
get_matrix() (matplotlib.projections.polar.PolarAffine method), 470
get_matrix() (matplotlib.projections.polar.PolarAxes.PolarAffine method), 471
get_matrix() (matplotlib.transforms.Affine2D method), 457
get_matrix() (matplotlib.transforms.BboxTransform method), 462
get_matrix() (matplotlib.transforms.BboxTransformFrom method), 463
get_matrix() (matplotlib.transforms.BboxTransformTo method), 463
get_matrix() (matplotlib.transforms.BlendedAffine2D method), 461
get_matrix() (matplotlib.transforms.CompositeAffine2D method), 462
get_matrix() (matplotlib.transforms.IdentityTransform method), 458
get_matrix() (matplotlib.transforms.ScaledTranslation method), 463
get_matrix() (matplotlib.transforms.Transform method), 452
get_mec() (matplotlib.lines.Line2D method), 1254
get_metrics() (matplotlib.mathtext.Fonts method), 1269
get_mfc() (matplotlib.lines.Line2D method), 1254
get_mfcalt() (matplotlib.lines.Line2D method), 1254
get_minimumdescent() (matplotlib.offsetbox.TextArea method), 1323
get_minor_formatter() (matplotlib.axis.Axis method), 954
get_minor_locator() (matplotlib.axis.Axis method), 954
get_minor_ticks() (matplotlib.axis.Axis method), 954
get_minorticklabels() (matplotlib.axis.Axis method), 954
get_minorticklines() (matplotlib.axis.Axis method), 954
get_minorticklocs() (matplotlib.axis.Axis method), 954
get_minpos() (matplotlib.axis.XAxis method), 959
get_minpos() (matplotlib.axis.YAxis method), 960
get_ms() (matplotlib.lines.Line2D method), 1254
get multilinebaseline() (matplotlib.offsetbox.TextArea method), 1323
get_mutation_aspect() (matplotlib.patches.FancyArrowPatch method), 1346
get_mutation_aspect() (matplotlib.patches.FancyBboxPatch method), 1349
get_mutation_scale() (matplotlib.patches.FancyArrowPatch method), 1347
get_mutation_scale() (matplotlib.patches.FancyBboxPatch method), 1349
get_name() (matplotlib.font_manager.FontProperties method), 1228
get_name() (matplotlib.text.Text method), 1578
get_name_char() (matplotlib.afm.AFM method), 744
get_navigate() (matplotlib.axes.Axes method), 856
get_navigate_mode() (matplotlib.axes.Axes method), 856
get_numpoints() (matplotlib.legend_handler.HandlerLineCollection method), 1248
get_numpoints() (matplotlib.legend_handler.HandlerRegularPolyCollection method), 1249
get_numpoints() (matplotlib.legend_handler.HandlerNpoints method), 1248
get_numpoints() (matplotlib.legend_handler.HandlerRegularPolyCollection method), 1249
get_numsides() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_numsides() (matplotlib.collections.RegularPolyCollection method), 1127
get_numsides() (matplotlib.collections.StarPolygonCollection method), 1138
get_offset() (matplotlib.offsetbox.AuxTransformBox method), 1318
get_offset() (matplotlib.offsetbox.DrawingArea method), 1320
get_offset() (matplotlib.offsetbox.OffsetBox method), 1321
get_offset() (matplotlib.offsetbox.OffsetImage method), 1322
get_offset() (matplotlib.offsetbox.TextArea method), 1323
get_offset() (matplotlib.ticker.FixedFormatter method), 1590
get_offset() (matplotlib.ticker.Formatter method), 1589
get_offset() (matplotlib.ticker.ScalarFormatter method), 1590
get_offset_position() (matplotlib.collections.AsteriskPolygonCollection method), 1008
get_offset_position() (matplotlib.collections.BrokenBarHCollection method), 1019
get_offset_position() (matplotlib.collections.CircleCollection method), 1030
get_offset_position() (matplotlib.collections.EllipseCollection method), 1051
get_offset_position() (matplotlib.collections.EventCollection method), 1063
get_offset_position() (matplotlib.collections.LineCollection method),
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_offset_position()</td>
<td>matplotlib.collections.PatchCollection</td>
<td>1085</td>
</tr>
<tr>
<td>get_offset_position()</td>
<td>matplotlib.collections.PathCollection</td>
<td>1095</td>
</tr>
<tr>
<td>get_offset_position()</td>
<td>matplotlib.collections.PolyCollection</td>
<td>1106</td>
</tr>
<tr>
<td>get_offset_position()</td>
<td>matplotlib.collections.QuadMesh</td>
<td>1117</td>
</tr>
<tr>
<td>get_offset_position()</td>
<td>matplotlib.collections.RegularPolyCollection</td>
<td>1127</td>
</tr>
<tr>
<td>get_offset_position()</td>
<td>matplotlib.collections.StarPolygonCollection</td>
<td>1139</td>
</tr>
<tr>
<td>get_offset_position()</td>
<td>matplotlib.collections.TriMesh</td>
<td>1149</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.AsteriskPolygonCollection</td>
<td>1008</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.BrokenBarHCollection</td>
<td>1019</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.CircleCollection</td>
<td>1030</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.EllipseCollection</td>
<td>1051</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.EventCollection</td>
<td>1063</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.LineCollection</td>
<td>1075</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.PatchCollection</td>
<td>1085</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.PathCollection</td>
<td>1095</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.PolyCollection</td>
<td>1106</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.QuadMesh</td>
<td>1117</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.RegularPolyCollection</td>
<td>1128</td>
</tr>
<tr>
<td>get_offsets()</td>
<td>matplotlib.collections.StarPolygonCollection</td>
<td>1139</td>
</tr>
<tr>
<td>Method</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>get_offsets() (matplotlib.collections.TriMesh method)</td>
<td>1149</td>
<td></td>
</tr>
<tr>
<td>get_orientation() (matplotlib.collections.EventCollection method)</td>
<td>1063</td>
<td></td>
</tr>
<tr>
<td>get_pad() (matplotlib.axis.Tick method)</td>
<td>958</td>
<td></td>
</tr>
<tr>
<td>get_pad() (mpl_toolkits.axes_grid1.axis_artist.AxesLabel method)</td>
<td>527,</td>
<td></td>
</tr>
<tr>
<td>get_pad() (mpl_toolkits.axes_grid1.axis_artist.AxesLabel method)</td>
<td>687</td>
<td></td>
</tr>
<tr>
<td>get_pad_pixels() (matplotlib.axis.Tick method)</td>
<td>958</td>
<td></td>
</tr>
<tr>
<td>get_pagecount() (matplotlib.backends.backend_pdf.PdfPages method)</td>
<td>984</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.Arrow method)</td>
<td>1329</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.Ellipse method)</td>
<td>1343</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.Patch method)</td>
<td>1352</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.Rectangle method)</td>
<td>1358</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.RegularPolygon method)</td>
<td>1360</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.Shadow method)</td>
<td>1361</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.Wedge method)</td>
<td>1362</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.patches.YAArrow method)</td>
<td>1364</td>
<td></td>
</tr>
<tr>
<td>get_patch_transform() (matplotlib.spines.Spine method)</td>
<td>1568</td>
<td></td>
</tr>
<tr>
<td>get_patch_collection_extents() (in module matplotlib.path)</td>
<td>1371</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.legend.AsteriskPolygonCollection method)</td>
<td>1008</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.BrokenBarHCollection method)</td>
<td>1019</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.CircleCollection method)</td>
<td>1030</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.Collection method)</td>
<td>1041</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.EllipseCollection method)</td>
<td>1051</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.EventCollection method)</td>
<td>1063</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.LineCollection method)</td>
<td>1075</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.PatchCollection method)</td>
<td>1085</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.PolyCollection method)</td>
<td>1095</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.collections.YCollection method)</td>
<td>1106</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.patches.Patch method)</td>
<td>1352</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.patches.Patch method)</td>
<td>1355</td>
<td></td>
</tr>
<tr>
<td>get_path() (matplotlib.patches.Polygon method)</td>
<td>1357</td>
<td></td>
</tr>
</tbody>
</table>
Index
Matplotlib, Release 1.4.3

plotlib.collections.LineCollection method), 1075
get_pickradius() (matplotlib.collections.PatchCollection method), 1085
get_pickradius() (matplotlib.collections.PathCollection method), 1095
get_pickradius() (matplotlib.collections.PolyCollection method), 1106
get_pickradius() (matplotlib.collections.QuadMesh method), 1117
get_pickradius() (matplotlib.collections.RegularPolyCollection method), 1128
get_pickradius() (matplotlib.collections.StarPolygonCollection method), 1139
get_pickradius() (matplotlib.collections.TriMesh method), 1149
get_pickradius() (matplotlib.collections.Line2D method), 1254
get_plot_commands() (in module matplotlib.pyplot), 1470
get_points() (matplotlib.transforms.Bbox method), 449
get_points() (matplotlib.transforms.TransformedBbox method), 450
get_position() (matplotlib.axes.Axes method), 856
get_position() (matplotlib.gridspec.SubplotSpec method), 1235
get_position() (matplotlib.spines.Spine method), 1568
get_position() (matplotlib.text.Text method), 1578
get_position() (matplotlib.text.TextWithDash method), 1584
get_position() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_position() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 522, 682
get_position_run() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_positions() (matplotlib.collections.EventCollection method), 1063
get_proj() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get_projection_class() (in module matplotlib.projections), 469
get_projection_class() (matplotlib.projections.ProjectionRegistry method), 469
get_projection_names() (in module matplotlib.projections), 469
get_prop_tup() (matplotlib.text.Text method), 1578
get_prop_tup() (matplotlib.text.TextWithDash method), 1584
get_proxy_renderer() (matplotlib.pathEffects.AbstractPathEffect method), 1373
get_radius() (matplotlib.patches.Circle method), 1337
get_rasterization_zorder() (matplotlib.axes.Axes method), 856
get_rasterized() (matplotlib.artist.Artist method), 758
get_rasterized() (matplotlib.axes.Axes method), 857
get_rasterized() (matplotlib.collections.AsteriskPolygonCollection method), 1009
get_rasterized() (matplotlib.collections.BrokenBarHCollection method), 1019
get_rasterized() (matplotlib.collections.CircleCollection method), 1030
get_rasterized() (matplotlib.collections.Collection method), 1041
get_rasterized() (matplotlib.collections.EllipseCollection method), 1052
get_rasterized() (matplotlib.collections.EventCollection method), 1063
get_rasterized() (matplotlib.collections.LineCollection method), 1075
get_rasterized() (matplotlib.collections.PatchCollection method), 1085
get_rasterized() (matplotlib.collections.PointCollection method), 1095
get_rasterized() (matplotlib.collections.QuadMesh method), 1117
get_rasterized() (matplotlib.collections.RegularPolyCollection method), 1128
get_rasterized() (matplotlib.collections.StarPolygonCollection method), 1139
get_rasterized() (matplotlib.collections.TriMesh method), 1149
get_rasterized() (matplotlib.collections.Line2D method), 1254
get_rasterized() (matplotlib.transforms.Bbox method), 449
get_rasterized() (matplotlib.transforms.TransformedBbox method), 450
get_position() (matplotlib.axes.Axes method), 856
get_position() (matplotlib.gridspec.SubplotSpec method), 1235
get_position() (matplotlib.spines.Spine method), 1568
get_position() (matplotlib.text.Text method), 1578
get_position() (matplotlib.text.TextWithDash method), 1584
get_position() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_position() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 522, 682
get_position_run() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 680
get_positions() (matplotlib.collections.EventCollection method), 1063
get_proj() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_size_inches()</td>
<td>matplotlib.figure.Figure method</td>
<td>1203</td>
</tr>
<tr>
<td>get_sized_alternatives_for_symbol()</td>
<td>matplotlib.math.text.BakomaFonts method</td>
<td>1268</td>
</tr>
<tr>
<td>get_sized_alternatives_for_symbol()</td>
<td>matplotlib.math.text.Fonts method</td>
<td>1270</td>
</tr>
<tr>
<td>get_sized_alternatives_for_symbol()</td>
<td>matplotlib.math.text.StixFonts method</td>
<td>1279</td>
</tr>
<tr>
<td>get_sized_alternatives_for_symbol()</td>
<td>matplotlib.math.text.UnicodeFonts method</td>
<td>1280</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1019</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.legend_handler.HandlerRegularPolygonCollection method</td>
<td>1249</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.artist.Artist method</td>
<td>758</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.axes.Axes method</td>
<td>857</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.backend_bases.GraphicsContextBase method</td>
<td>971</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1020</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.Collection method</td>
<td>1041</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EllipseCollection method</td>
<td>1052</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EventCollection method</td>
<td>1063</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.LineCollection method</td>
<td>1075</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PatchCollection method</td>
<td>1085</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.QuadMesh method</td>
<td>1117</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.TriMesh method</td>
<td>1149</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1249</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.artist.Artist method</td>
<td>758</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.axes.Axes method</td>
<td>857</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.backend_bases.GraphicsContextBase method</td>
<td>971</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1020</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.Collection method</td>
<td>1041</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EllipseCollection method</td>
<td>1052</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EventCollection method</td>
<td>1063</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.LineCollection method</td>
<td>1075</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PatchCollection method</td>
<td>1085</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.QuadMesh method</td>
<td>1117</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.TriMesh method</td>
<td>1149</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1249</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.artist.Artist method</td>
<td>758</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.axes.Axes method</td>
<td>857</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.backend_bases.GraphicsContextBase method</td>
<td>971</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1020</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.Collection method</td>
<td>1041</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EllipseCollection method</td>
<td>1052</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EventCollection method</td>
<td>1063</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.LineCollection method</td>
<td>1075</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PatchCollection method</td>
<td>1085</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.QuadMesh method</td>
<td>1117</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.TriMesh method</td>
<td>1149</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1249</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.artist.Artist method</td>
<td>758</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.axes.Axes method</td>
<td>857</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.backend_bases.GraphicsContextBase method</td>
<td>971</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1020</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.Collection method</td>
<td>1041</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EllipseCollection method</td>
<td>1052</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EventCollection method</td>
<td>1063</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.LineCollection method</td>
<td>1075</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PatchCollection method</td>
<td>1085</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.QuadMesh method</td>
<td>1117</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.TriMesh method</td>
<td>1149</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1249</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.artist.Artist method</td>
<td>758</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.axes.Axes method</td>
<td>857</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.backend_bases.GraphicsContextBase method</td>
<td>971</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1020</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.Collection method</td>
<td>1041</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EllipseCollection method</td>
<td>1052</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EventCollection method</td>
<td>1063</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.LineCollection method</td>
<td>1075</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PatchCollection method</td>
<td>1085</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.QuadMesh method</td>
<td>1117</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.TriMesh method</td>
<td>1149</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1249</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.artist.Artist method</td>
<td>758</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.axes.Axes method</td>
<td>857</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.backend_bases.GraphicsContextBase method</td>
<td>971</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1020</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.Collection method</td>
<td>1041</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EllipseCollection method</td>
<td>1052</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EventCollection method</td>
<td>1063</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.LineCollection method</td>
<td>1075</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PatchCollection method</td>
<td>1085</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.QuadMesh method</td>
<td>1117</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.TriMesh method</td>
<td>1149</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1249</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.artist.Artist method</td>
<td>758</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.axes.Axes method</td>
<td>857</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.backend_bases.GraphicsContextBase method</td>
<td>971</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method</td>
<td>1009</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.BrokenBarHCollection method</td>
<td>1020</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.CircleCollection method</td>
<td>1030</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.Collection method</td>
<td>1041</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EllipseCollection method</td>
<td>1052</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.EventCollection method</td>
<td>1063</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.LineCollection method</td>
<td>1075</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PatchCollection method</td>
<td>1085</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PathCollection method</td>
<td>1096</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.PolyCollection method</td>
<td>1106</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.QuadMesh method</td>
<td>1117</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.RegularPolyCollection method</td>
<td>1128</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.StarPolygonCollection method</td>
<td>1139</td>
</tr>
<tr>
<td>get_sizes()</td>
<td>matplotlib.collections.TriMesh method</td>
<td>1149</td>
</tr>
</tbody>
</table>
get_snap (matplotlib.collections.EllipseCollection method), 1052
get_snap (matplotlib.collections.EventCollection method), 1064
get_snap (matplotlib.collections.LineCollection method), 1075
get_snap (matplotlib.collections.PatchCollection method), 1085
get_snap (matplotlib.collections.PathCollection method), 1096
get_snap (matplotlib.collections.TriMesh method), 1107
get_snap (matplotlib.collections.QuadMesh method), 1117
get_snap (matplotlib.collections.RegularPolyCollection method), 1128
get_snap (matplotlib.collections.StarPolygonCollection method), 1139
get_snap (matplotlib.collections.PolyCollection method), 1150
get_snap_threshold (matplotlib.markers.MarkerStyle method), 1234
get_subplot_params (matplotlib.gridspec.GridSpec method), 1244
get_subplotspec (mpl_toolkits.axes_grid.areas_divider.AxesLocator method), 522, 682
get_subplotspec (mpl_toolkits.axes_grid.areas_divider.SubplotDivider method), 522, 682
get_subplotspec_list (in module matplotlib.tight_layout), 1597
get_supported_filetypes (matplotlib.backend_bases.FigureCanvasBase class method), 965
get_supported_filetypes_grouped (matplotlib.backend_bases.FigureCanvasBase class method), 965
get_test_data (in module mpl_toolkits.mplot3d.axes3d), 576, 633
get_textmanager (matplotlib.backend_bases.RendererBase method), 979
get_text (matplotlib.offsetbox.TextArea method), 1324
get_text (matplotlib.text.Text method), 1579
get_text_heights (matplotlib.axis.XAxis method), 959
get_text_width_height_descent (matplotlib.backend_bases.RendererBase method), 979
get_text_widths (matplotlib.axis.YAxis method), 961
get_texts (matplotlib.legend.Legend method), 1246
get_texts_widths_heights_descents (mpl_toolkits.axes_grid.axis_artist.TickLabels method), 528, 687
get_theta_direction (matplotlib.projections.polar.PolarAxes method), 472
get_theta_offset (matplotlib.projections.polar.PolarAxes method), 472
get_tick_out (mpl_toolkits.axes_grid.axis_artist.Ticks method), 527, 686
get_tick_positions (mpl_toolkits.mplot3d.axes3d.Axes3D method), 576, 634
get_ticklabel_extents (matplotlib.axis.Axis method), 954
get_ticklabels (matplotlib.axis.Axis method), 954

Index 2575
get_ticklines() (matplotlib.axis.Axis method), 955
get_ticklocs() (matplotlib.axis.Axis method), 955
get_ticks_position() (matplotlib.axis.XAxis method), 959
get_ticks_position() (matplotlib.axis.YAxis method), 961
get_ticksize() (mpl_toolkits.axes_grid.axis_artist.Ticks method), 527, 686
get_tick_layout() (matplotlib.figure.Figure method), 1203
get_tick_layout_figure() (in module matplotlib.tight_layout), 1597
get_tightbbox() (matplotlib.axes.Axes method), 857
get_tightbbox() (matplotlib.axis.Axis method), 955
get_tightbbox() (matplotlib.figure.Figure method), 1203
get_tightbbox() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 525, 685
get_tightbbox() (mpl_toolkits.mplot3d.axis3d.Axis method), 576, 634
get_title() (matplotlib.axes.Axes method), 857
get_title() (matplotlib.legend.Legend method), 1246
get_topmost_subplotspec() (matplotlib.gridspec.GridSpecFromSubplotSpec method), 1234
get_topmost_subplotspec() (matplotlib.gridspec.SubplotSpec method), 1235
get_transform() (matplotlib.artist.Artist method), 758
get_transform() (matplotlib.axes.Axes method), 857
get_transform() (matplotlib.axis.Axis method), 955
get_transform() (matplotlib.collections.AsteriskPolygonCollection method), 1009
get_transform() (matplotlib.collections.BrokenBarHCollection method), 1020
get_transform() (matplotlib.collections.CircleCollection method), 1031
get_transform() (matplotlib.collections.Collection method), 1042
get_transform() (matplotlib.collections.EllipseCollection method), 1052
get_transform() (matplotlib.collections.LineCollection method), 1075
get_transform() (matplotlib.collections.PatchCollection method), 1086
get_transform() (matplotlib.collections.PolyCollection method), 1096
get_transform() (matplotlib.collections.Patch method), 1107
get_transform() (matplotlib.collections.QuadMesh method), 1118
get_transform() (matplotlib.collections.RegularPolyCollection method), 1128
get_transform() (matplotlib.collections.StarPolygonCollection method), 1139
get_transform() (matplotlib.collections.TriMesh method), 1150
get_transform() (matplotlib.markers.MarkerStyle method), 1263
get_transform() (matplotlib.offsetbox.AuxTransformBox method), 1318
get_transform() (matplotlib.offsetbox.DrawingArea method), 1320
get_transform() (matplotlib.patches.Patch method), 1352
get_transform() (matplotlib.scale.LinearScale method), 467
get_transform() (matplotlib.scale.LogScale method), 467
get_transform() (matplotlib.scale.ScaleBase method), 468
get_transform() (matplotlib.scale.SymmetricalLogScale method), 468
get_transform() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 525, 685
get_transformed_clip_path_and_affine() (matplotlib.artist.Artist method), 758
get_transformed_clip_path_and_affine() (matplotlib.artist.Artist method), 857
get_transformed_clip_path_and_affine() (matplotlib.collections.AsteriskPolygonCollection method), 1009
get_transformed_points_and_affine() (matplotlib.transforms.TransformedPath method), 463
get_transforms() (matplotlib.collections.AsteriskPolygonCollection method), 1009
get_transforms() (matplotlib.collections.BrokenBarHCollection method), 1020
get_transforms() (matplotlib.collections.CircleCollection method), 1031
get_transforms() (matplotlib.collections.Collection method), 1042
get_transforms() (matplotlib.collections.EllipseCollection method), 1052
get_transforms() (matplotlib.collections.EventCollection method), 1064
get_transforms() (matplotlib.collections.LineCollection method), 1075
get_transforms() (matplotlib.collections.PatchCollection method), 1086
get_transforms() (matplotlib.collections.PathCollection method), 1096
get_transforms() (matplotlib.collections.PolyCollection method), 1107
get_transforms() (matplotlib.collections.QuadMesh method), 1118
get_transforms() (matplotlib.collections.RegularPolyCollection method), 1129
get_transforms() (matplotlib.collections.StarPolygonCollection method), 1140
get_transforms() (matplotlib.collections.TriMesh method), 1150
get_transformed_clip_path_and_affine() (matplotlib.tri.Triangulation method), 1600
get_trifinder() (matplotlib.tri.Triangulation method), 1600
get_underline_thickness() (matplotlib.afm.AFM method), 744
get_underline_thickness() (matplotlib.mathtext.Fonts method), 1270
get_underline_thickness() (matplotlib.mathtext.StandardPsFonts method), 1278
get_transforms() (matplotlib.collections.BrokenBarHCollection method), 1020
get_transforms() (matplotlib.collections.CircleCollection method), 1031
get_transforms() (matplotlib.collections.Collection method), 1042
get_transforms() (matplotlib.collections.EllipseCollection method), 1052
get_transforms() (matplotlib.collections.EventCollection method), 1064
get_transforms() (matplotlib.collections.LineCollection method), 1075
get_transforms() (matplotlib.collections.PatchCollection method), 1086
get_transforms() (matplotlib.collections.PathCollection method), 1096
get_transforms() (matplotlib.collections.PolyCollection method), 1107
get_transforms() (matplotlib.collections.QuadMesh method), 1118
get_transforms() (matplotlib.collections.RegularPolyCollection method), 1129
get_transforms() (matplotlib.collections.StarPolygonCollection method), 1140
get_transforms() (matplotlib.collections.TriMesh method), 1150
get_trifinder() (matplotlib.tri.Triangulation method), 1600
get_underline_thickness() (matplotlib.afm.AFM method), 744
get_underline_thickness() (matplotlib.mathtext.Fonts method), 1270
get_underline_thickness() (matplotlib.mathtext.StandardPsFonts method), 1278
get_transforms() (matplotlib.collections.AsteriskPolygonCollection method), 1009
get_transforms() (matplotlib.collections.BrokenBarHCollection method), 1020
get_transforms() (matplotlib.collections.CircleCollection method), 1031
get_transforms() (matplotlib.collections.Collection method), 1042
get_transforms() (matplotlib.collections.EllipseCollection method), 1052
get_transforms() (matplotlib.collections.EventCollection method), 1064
get_transforms() (matplotlib.collections.LineCollection method), 1075
get_transforms() (matplotlib.collections.PatchCollection method), 1086
get_transforms() (matplotlib.collections.PathCollection method), 1096
get_transforms() (matplotlib.collections.PolyCollection method), 1107
get_transforms() (matplotlib.collections.QuadMesh method), 1118
get_transforms() (matplotlib.collections.RegularPolyCollection method), 1129
get_transforms() (matplotlib.collections.StarPolygonCollection method), 1140
get_transforms() (matplotlib.collections.TriMesh method), 1150
get_trifinder() (matplotlib.tri.Triangulation method), 1600
get_underline_thickness() (matplotlib.afm.AFM method), 744
get_underline_thickness() (matplotlib.mathtext.Fonts method), 1270
get_underline_thickness() (matplotlib.mathtext.StandardPsFonts method), 1278
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_unicode_index()</td>
<td>matplotlib.mathtext</td>
<td>1279</td>
</tr>
<tr>
<td>get_unit()</td>
<td>matplotlib.text.OffsetFrom</td>
<td>1576</td>
</tr>
<tr>
<td>get_unit_generic()</td>
<td>matplotlib.dates.RRuleLocator</td>
<td>1179</td>
</tr>
<tr>
<td>get_units()</td>
<td>matplotlib.axis.Axis</td>
<td>955</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.axes.Artist</td>
<td>758</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.axes.Axes</td>
<td>858</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.backend_bases</td>
<td>971</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections</td>
<td>1009</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.BrokenBarHCollection</td>
<td>1010</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.CircleCollection</td>
<td>1031</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.Collection</td>
<td>1042</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.EllipseCollection</td>
<td>1052</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.EventCollection</td>
<td>1064</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.LineCollection</td>
<td>1075</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.PatchCollection</td>
<td>1086</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.PathCollection</td>
<td>1096</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.PolyCollection</td>
<td>1107</td>
</tr>
<tr>
<td>get_url()</td>
<td>matplotlib.collections.QuadMesh</td>
<td>1118</td>
</tr>
<tr>
<td>get_used_characters()</td>
<td>matplotlib.mathtext.Fonts</td>
<td>1270</td>
</tr>
<tr>
<td>get_useLocale()</td>
<td>matplotlib.ticker.ScalarFormatter</td>
<td>1591</td>
</tr>
<tr>
<td>get_useOffset()</td>
<td>matplotlib.ticker.ScalarFormatter</td>
<td>1591</td>
</tr>
<tr>
<td>get_va()</td>
<td>matplotlib.text.Text</td>
<td>1579</td>
</tr>
<tr>
<td>get_valid_values()</td>
<td>matplotlib.artist.ArtistInspector</td>
<td>763</td>
</tr>
<tr>
<td>get_variant()</td>
<td>matplotlib.font_manager.FontProperties</td>
<td>1228</td>
</tr>
<tr>
<td>get_variant()</td>
<td>matplotlib.text.Text</td>
<td>1579</td>
</tr>
<tr>
<td>get_vector()</td>
<td>mpl_toolkits.mplot3d</td>
<td>580</td>
</tr>
<tr>
<td>get_vertical()</td>
<td>mpl_toolkits.axes_grid</td>
<td>520</td>
</tr>
<tr>
<td>get_vertical_sizes()</td>
<td>mpl_toolkits.axes_grid</td>
<td>520</td>
</tr>
<tr>
<td>get_vertical_stem_width()</td>
<td>matplotlib.afm.AFM</td>
<td>744</td>
</tr>
<tr>
<td>get_verticalalignment()</td>
<td>matplotlib.text.Text</td>
<td>1579</td>
</tr>
<tr>
<td>get_verts()</td>
<td>matplotlib.patches.Patch</td>
<td>1353</td>
</tr>
<tr>
<td>get_view_interval()</td>
<td>matplotlib.axis.Axes</td>
<td>955</td>
</tr>
<tr>
<td>get_view_interval()</td>
<td>matplotlib.axis.Tick</td>
<td>958</td>
</tr>
</tbody>
</table>
get_view_interval() (matplotlib.axis.XAxis method), 959
get_view_interval() (matplotlib.axis.XTick method), 960
get_view_interval() (matplotlib.axis.YAxis method), 961
get_view_interval() (matplotlib.axis.YTick method), 962
get_view_interval() (mpl_toolkits.mplot3d.axis3d.Axis method), 576, 634
get_visible() (matplotlib.artist.Artist method), 759
get_visible() (matplotlib.axes.Axes method), 858
get_visible() (matplotlib.collections.AsteriskPolygonCollection method), 1010
get_visible() (matplotlib.collections.BrokenBarHCollection method), 1020
get_visible() (matplotlib.collections.CircleCollection method), 1031
get_visible() (matplotlib.collections.Collection method), 1042
get_visible() (matplotlib.collections.EllipseCollection method), 1052
get_visible() (matplotlib.collections.EventCollection method), 1064
get_visible() (matplotlib.collections.LineCollection method), 1076
get_visible() (matplotlib.collections.PatchCollection method), 1086
get_visible() (matplotlib.collections.PathCollection method), 1096
get_visible() (matplotlib.collections.PolyCollection method), 1107
get_visible() (matplotlib.collections.QuadMesh method), 1118
get_visible() (matplotlib.collections.RegularPolyCollection method), 1129
get_visible() (matplotlib.collections.StarPolygonCollection method), 1140
get_visible() (matplotlib.collections.TriMesh method), 1150
get_visible_children() (matplotlib.offsetbox.OffsetBox method), 1321
get_vsize_hsize() (mpl_toolkits.axes_grid.axes_divider.Divider method), 520, 681
get_w_lims() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get_weight() (matplotlib.afm.AFM method), 744
get_weight() (matplotlib.font_manager.FontProperties method), 1228
get_weight() (matplotlib.text.Text method), 1579
get_width() (matplotlib.patches.FancyBboxPatch method), 1349
get_width() (matplotlib.patches.Rectangle method), 1358
get_width_char() (matplotlib.afm.AFM method), 744
get_width_from_char_name() (matplotlib.afm.AFM method), 744
get_width_height() (matplotlib.backend_bases.FigureCanvasBase method), 965
get_width_ratios() (matplotlib.gridspec.GridSpecBase method), 1234
get_window_extent() (matplotlib.artist.Artist method), 759
get_window_extent() (matplotlib.axes.Axes method), 858
get_window_extent() (matplotlib.collections.AsteriskPolygonCollection method), 1010
get_window_extent() (matplotlib.collections.BrokenBarHCollection method), 1020
get_window_extent() (matplotlib.collections.CircleCollection method), 1031
get_window_extent() (matplotlib.collections.Collection method), 1042
get_window_extent() (matplotlib.collections.EllipseCollection method), 1052
get_window_extent() (matplotlib.collections.EventCollection method), 1064
get_window_extent() (matplotlib.collections.LineCollection method), 1076
get_window_extent() (matplotlib.collections.PatchCollection method), 1086
get_window_extent() (matplotlib.collections.PathCollection method), 1096
get_window_extent() (matplotlib.collections.PolyCollection method), 1107
get_window_extent() (matplotlib.collections.QuadMesh method), 1118
get_window_extent() (matplotlib.collections.RegularPolyCollection method), 1129
get_window_extent() (matplotlib.collections.StarPolygonCollection method), 1140
get_window_extent() (matplotlib.collections.TriMesh method), 1150
get_window_extent() (matplotlib.collections.EllipseCollection method), 1052
get_window_extent() (matplotlib.collections.EventCollection method), 1064
get_window_extent() (matplotlib.collections.LineCollection method), 1076
get_window_extent() (matplotlib.collections.PatchCollection method), 1086
get_window_extent() (matplotlib.collections.PathCollection method), 1096
get_window_extent() (matplotlib.collections.PolyCollection method), 1107
get_window_extent() (matplotlib.collections.QuadMesh method), 1118
get_window_extent() (matplotlib.collections.RegularPolyCollection method), 1129
get_window_extent() (matplotlib.collections.StarPolygonCollection method), 1140
get_window_extent() (matplotlib.collections.TriMesh method), 1150
get_window_extent() (matplotlib.figure.Figure method), 1203
get_window_extent() (matplotlib.image.BboxImage method), 1238
get_window_extent() (matplotlib.legend.Legend method), 1246
get_window_extent() (matplotlib.lines.Line2D method), 1254
get_window_extent() (matplotlib.offsetbox.ArchimedeanOffsetBox method), 1316
get_window_extent() (matplotlib.offsetbox.AuxTransformBox method), 1318
get_window_extent() (matplotlib.offsetbox.DrawingArea method), 1320
get_window_extent() (matplotlib.offsetbox.OffsetBox method), 1321
get_window_extent() (matplotlib.offsetbox.OffsetImage method), 1322
get_window_extent() (matplotlib.offsetbox.TextArea method), 1324
get_window_title() (matplotlib.backend_bases.FigureCanvasBase method), 956
get_window_title() (matplotlib.backend_bases.FigureManagerBase method), 959
get_x() (matplotlib.patches.FancyBboxPatch method), 1349
get_x() (matplotlib.patches.Rectangle method), 1358
get_xaxis() (matplotlib.axes.Axes method), 858
get_xaxis_text1_transform() (matplotlib.axes.Axes method), 858
get_xaxis_text2_transform() (matplotlib.axes.Axes method), 858
get_xaxis_transform() (matplotlib.axes.Axes method), 858
get_xbound() (matplotlib.axes.Axes method), 858
get_xdata() (matplotlib.legend_handler.HandlerNpoints method), 1248
get_xdata() (matplotlib.legend_handler.HandlerNpoints method), 1248
get_xgridlines() (matplotlib.axes.Axes method), 859
get_xheight() (matplotlib.afm.AFM method), 744
get_xheight() (matplotlib.mathtext.Fonts method), 1270
get_xheight() (matplotlib.mathtext.StandardPsFonts method), 1278
get_xheight() (matplotlib.mathtext.TrueTypeFonts method), 1279
get_xlabel() (matplotlib.axes.Axes method), 859
get_xlim() (matplotlib.axes.Axes method), 859
get_xlim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get_xlim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get_xmajorticklabels() (matplotlib.axes.Axes method), 859
get_xminorticklabels() (matplotlib.axes.Axes method), 859
get_xscale() (matplotlib.axes.Axes method), 859
get_xticklabels() (matplotlib.axes.Axes method), 859
get_xticklabels() (matplotlib.axes.Axes method), 859
get_window_extent() (matplotlib.patches.Patch method), 1353
get_window_extent() (matplotlib.text.Annotation method), 1576
get_window_extent() (matplotlib.text.Text method), 1579
get_window_extent() (matplotlib.text.TextWithDash method), 1584
get_window_title() (matplotlib.backend_bases.FigureCanvasBase method), 965
get_window_title() (matplotlib.backend_bases.FigureManagerBase method), 969
get_x() (matplotlib.patches.Rectangle method), 1358
get_xaxis() (matplotlib.axes.Axes method), 858
get_xaxis_text1_transform() (matplotlib.axes.Axes method), 858
get_xaxis_text2_transform() (matplotlib.axes.Axes method), 858
get_xaxis_transform() (matplotlib.axes.Axes method), 858
get_xbound() (matplotlib.axes.Axes method), 858
get_xdata() (matplotlib.legend_handler.HandlerNpoints method), 1248
get_xdata() (matplotlib.legend_handler.HandlerNpoints method), 1248
get_xgridlines() (matplotlib.axes.Axes method), 859
get_xheight() (matplotlib.afm.AFM method), 744
get_xheight() (matplotlib.mathtext.Fonts method), 1270
get_xheight() (matplotlib.mathtext.StandardPsFonts method), 1278
get_xheight() (matplotlib.mathtext.TrueTypeFonts method), 1279
get_xlabel() (matplotlib.axes.Axes method), 859
get_xlim() (matplotlib.axes.Axes method), 859
get_xlim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get_xlim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get_xmajorticklabels() (matplotlib.axes.Axes method), 859
get_xminorticklabels() (matplotlib.axes.Axes method), 859
get_xscale() (matplotlib.axes.Axes method), 859
get_xticklabels() (matplotlib.axes.Axes method), 859
get_xticklines() (matplotlib.axes.Axes method), 859
get_xticks() (matplotlib.axes.Axes method), 859
get_xy() (matplotlib.patches.Polygon method), 1357
get_xy() (matplotlib.patches.Rectangle method), 1358
get_xydata() (matplotlib.lines.Line2D method), 1254
get_xyz_where() (in module matplotlib.mlab), 1300
get_y() (matplotlib.patches.FancyBboxPatch method), 1349
get_y() (matplotlib.patches.Rectangle method), 1358
get_yaxis() (matplotlib.axes.Axes method), 859
get_yaxis_text1_transform() (matplotlib.axes.Axes method), 859
get_yaxis_text2_transform() (matplotlib.axes.Axes method), 859
get_yaxis_transform() (matplotlib.axes.Axes method), 860
get_ybound() (matplotlib.axes.Axes method), 860
get_ydata() (matplotlib.legend_handler.HandlerNpointsYoff sets method), 1248
get_ydata() (matplotlib.legend_handler.HandlerStem method), 1249
get_ydata() (matplotlib.lines.Line2D method), 1254
get_ygridlines() (matplotlib.axes.Axes method), 860
get_ylabel() (matplotlib.axes.Axes method), 860
get ylim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get ylim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get ylim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 559, 618
get_ymajorticklabels() (matplotlib.axes.Axes method), 860
get_ymajorticklabels() (matplotlib.patches.Axes3D method), 560, 618
get_yminorticklabels() (matplotlib.patches.Axes3D method), 560, 618
get_zoom() (matplotlib.offsetbox.OffsetImage method), 1322
get_zorder() (matplotlib.artist.Artist method), 759
get_zorder() (matplotlib.axes.Axes method), 861
get_zorder() (matplotlib.collections.AsteriskPolygonCollection method), 1010
get_zorder() (matplotlib.collections.BrokenBarHCollection method), 1020
get_zorder() (matplotlib.collections.CircleCollection method), 1031
get_zorder() (matplotlib.collections.Collection method), 1042
get_zorder() (matplotlib.collections.EventCollection method), 1052
get_zorder() (matplotlib.collections.LineCollection method), 1064
get_zorder() (matplotlib.collections.LineCollection method), 1076
get_zorder() (matplotlib.collections.PathCollection method), 1086
get_zorder() (matplotlib.collections.PathCollection method), 1096
get_zorder() (matplotlib.collections.PatchCollection method), 1107
get_zorder() (matplotlib.collections.QuadMesh method), 1118
get_zorder() (matplotlib.collections.RegularPolyCollection method), 1129
get_zorder() (matplotlib.collections.StarPolygonCollection method), 1140
get_zorder() (matplotlib.collections.TriMesh method), 1150
get_zscale() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 560, 619
get_zticklabels() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 560, 619
get_zticklabels() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 560, 619
get_zticks() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 560, 619
get_zticks() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 560, 619
getp() (in module matplotlib.artist), 764
getpoints() (matplotlib.patches.YAArrow method), 1364
GetRealpathAndStat (class in matplotlib.cbook), 988
ginput() (in module matplotlib.pyplot), 1470
ginput() (matplotlib.figure.Figure method), 1203
Glue (class in matplotlib.mathtext), 1270
GlueSpec (class in matplotlib.mathtext), 1270
grab_frame() (matplotlib.animation.FileMovieWriter method), 749
grab_frame() (matplotlib.animation.MovieWriter method), 752
grab_mouse() (matplotlib.backend_bases.FigureCanvasBase method), 965
gradient() (matplotlib.tri.CubicTriInterpolator method), 1603
gradient() (matplotlib.tri.LinearTriInterpolator method), 1601
GraphicsContextBase (class in matplotlib.backend_bases), 970
gray() (in module matplotlib.pyplot), 1471
Grid (class in mpl_toolkits.axes_grid.axes_grid), 524, 683
grid() (in module matplotlib.pyplot), 1471
grid() (matplotlib.axes.Axes method), 861
grid() (matplotlib.axis.Axis method), 955
grid() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 560, 619
griddata() (in module matplotlib.mlab), 1300
GridSpec (class in matplotlib.gridspec), 1233
GridSpecBase (class in matplotlib.gridspec), 1234
GridSpecFromSubplotSpec (class in matplotlib.gridspec), 1234
group() (matplotlib.mathtext.Parser method), 1276
Grouper (class in matplotlib.cbook), 988
grow() (matplotlib.mathtext.Accent method), 1267
grow() (matplotlib.mathtext.Box method), 1268
grow() (matplotlib.mathtext.Char method), 1268
grow() (matplotlib.mathtext.Glue method), 1270
grow() (matplotlib.mathtext.Kern method), 1271
grow() (matplotlib.mathtext.List method), 1272
grow() (matplotlib.mathtext.Node method), 1275
GTK, 2529

HAND (matplotlib.backend_bases.Cursors attribute), 963
HandlerBase (class in matplotlib.legend_handler), 1247
HandlerCircleCollection (class in matplotlib.legend_handler), 1247
HandlerErrorbar (class in matplotlib.legend_handler), 1247
HandlerLine2D (class in matplotlib.legend_handler), 1248
HandlerLineCollection (class in matplotlib.legend_handler), 1248
HandlerNpoints (class in matplotlib.legend_handler), 1248
HandlerNpointsYoffsets (class in matplotlib.legend_handler), 1248
HandlerPatch (class in matplotlib.legend_handler), 1248
HandlerPathCollection (class in matplotlib.legend_handler), 1249
HandlerRegularPolyCollection (class in matplotlib.legend_handler), 1249
HandlerStem (class in matplotlib.legend_handler), 1249
HandlerTuple (class in matplotlib.legend_handler), 1249
has_data() (matplotlib.axes.Axes method), 862
has_inverse (matplotlib.transforms.Transform attribute), 452
has_nonfinite (matplotlib.path.Path attribute), 1368
hatch() (matplotlib.path.Path class method), 1368
have_units() (matplotlib.artist.Artist method), 759
have_units() (matplotlib.axes.Axes method), 862
have_units() (matplotlib.axes.Axis method), 955
have_units() (matplotlib.collections.AsteriskPolygonCollection method), 1010
have_units() (matplotlib.collections.BrokenBarHCollection method), 1020
have_units() (matplotlib.collections.CircleCollection method), 1031
have_units() (matplotlib.collections.Collection method), 1042
have_units() (matplotlib.collections.EllipseCollection method), 1052
have_units() (matplotlib.collections.EventCollection method), 1064
Matplotlib, Release 1.4.3

method), 460
inverted() (matplotlib.transforms.CompositeGenericTransform
method), 461
inverted() (matplotlib.transforms.IdentityTransform
method), 458
inverted() (matplotlib.transforms.Transform
method), 452
InvertedPolarTransform (class in matplotlib.projections.polar), 469
ioff() (in module matplotlib.pyplot), 1484
ion() (in module matplotlib.pyplot), 1484
is_alias() (matplotlib.artist.ArtistInspector method), 763
is_available() (matplotlib.animation.MovieWriterRegistry
method), 753
is_closed_polygon() (in module matplotlib.mlab), 1301
is_color_like() (in module matplotlib.colors), 1172
is_dashed() (matplotlib.lines.Line2D method), 1254
is_dropsub() (matplotlib.mathtext.Parser method), 1276
is_figure_set() (matplotlib.artist.ArtistInspector method), 759
is_figure_set() (matplotlib.axes.Axes method), 874
is_figure_set() (matplotlib.collections.AsteriskPolygonCollection
method), 1010
is_figure_set() (matplotlib.collections.BrokenBarHCollection
method), 1020
is_figure_set() (matplotlib.collections.CircleCollection
method), 1031
is_figure_set() (matplotlib.collections.Collection
method), 1042
is_figure_set() (matplotlib.collections.EllipseCollection
method), 1053
is_figure_set() (matplotlib.collections.EventCollection
method), 1064
is_figure_set() (matplotlib.collections.LineCollection
method), 1076
is_figure_set() (matplotlib.collections.PatchCollection
method), 1086
is_figure_set() (matplotlib.collections.PathCollection
method), 1097
is_figure_set() (matplotlib.collections.PolyCollection
method), 1107
is_figure_set() (matplotlib.collections.QuadMesh
method), 1118
is_figure_set() (matplotlib.collections.RegularPolyCollection
method), 1129
is_figure_set() (matplotlib.collections.StarPolygonCollection
method), 1140
is_figure_set() (matplotlib.collections.TriMesh
method), 1150
is_filled() (matplotlib.markers.MarkerStyle method), 1263
is_frame_like() (matplotlib.spines.Spine method), 1568
is_gray() (matplotlib.colors.Colormap method), 1167
is_horizontal() (matplotlib.collections.EventCollection
method), 1064
is_math_text() (in module matplotlib.cbook), 995
is_math_text() (matplotlib.text.Text static method), 1579
is_missing() (matplotlib.cbook.converter method), 993
is_numlike() (in module matplotlib.cbook), 995
is_numlike() (matplotlib.units.ConversionInterface
static method), 1614
is_opentype_cff_font() (in module matplotlib.font_manager), 1230
is_overunder() (matplotlib.mathtext.Parser method), 1276
is_saving() (matplotlib.backend_bases.FigureCanvasBase
method), 965
is_scalar() (in module matplotlib.cbook), 995
is_scalar_or_string() (in module matplotlib.cbook), 995
is_separable (matplotlib.transforms.Transform
attribute), 452
is_sequence_of_strings() (in module matplotlib.cbook), 995
is_slanted() (matplotlib.mathtext.Char method), 1268
is_slanted() (matplotlib.mathtext.Parser method), 2585
is_string_like() (in module matplotlib.cbook), 996
is_transform_set() (matplotlib.artist.Artist method), 759
is_transform_set() (matplotlib.axes.Axes method), 874
is_transform_set() (matplotlib.collections.AsteriskPolygonCollection method), 1010
is_transform_set() (matplotlib.collections.BrokenBarHCollection method), 1021
is_transform_set() (matplotlib.collections.CircleCollection method), 1031
is_transform_set() (matplotlib.collections.Collection method), 1042
is_transform_set() (matplotlib.collections.EllipseCollection method), 1053
is_transform_set() (matplotlib.collections.EventCollection method), 1064
is_transform_set() (matplotlib.collections.LineCollection method), 1076
is_transform_set() (matplotlib.collections.PatchCollection method), 1086
is_transform_set() (matplotlib.collections.PathCollection method), 1097
is_transform_set() (matplotlib.collections.PolyCollection method), 1107
is_transform_set() (matplotlib.collections.QuadMesh method), 1118
is_transform_set() (matplotlib.collections.RegularPolygonCollection method), 1129
is_transform_set() (matplotlib.collections.StarPolygonCollection method), 1140
is_transform_set() (matplotlib.collections.TriMesh method), 1150
is_unit() (matplotlib.transforms.BboxBase method), 447
is_writable_file_like() (in module matplotlib.cbook), 996
isAvailable() (matplotlib.animation.MovieWriter class method), 753
iscolor() (in module mpl_toolkits.mplot3d.art3d), 581, 638
ishold() (in module matplotlib.pyplot), 1484
ishold() (matplotlib.axes.Axes method), 874
isinteractive() (in module matplotlib.pyplot), 1484
isowner() (matplotlib.widgets.LockDraw method), 1619
ispower2() (in module matplotlib.mlab), 1301
issubclass_safe() (in module matplotlib.mlab), 996
isvector() (in module matplotlib.pyplot), 1484
iter_segments() (matplotlib.path.Path method), 1368
iter_ticks() (matplotlib.axis.Axis method), 955
iterable() (in module matplotlib.pyplot), 1484
J
jet() (in module matplotlib.pyplot), 1484
join() (matplotlib.cbook.Grouper method), 989
joined() (matplotlib.cbook.Grouper method), 989
JPG, 2529
juggle_axes() (in module mpl_toolkits.mplot3d.art3d), 581, 638
K
Kern (class in matplotlib.mathtext), 1271
kern() (matplotlib.mathtext.Hlist method), 1271
key_press() (matplotlib.backend_bases.FigureManagerBase method), 969
key_press_event() (matplotlib.backend_bases.FigureCanvasBase method), 965
key_press_handler() (in module matplotlib.backend_bases), 981
key_press_handler_id (matplotlib.backend_bases.FigureManagerBase attribute), 969
key_release_event() (matplotlib.backend_bases.FigureCanvasBase method), 965
KeyEvent (class in matplotlib.backend_bases), 973
kwdoc() (in module matplotlib.artist), 764
L
11norm() (in module matplotlib.mlab), 1301
l2norm() (in module matplotlib.mlab), 1301
Index
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>longest_contiguous_ones()</td>
<td>matplotlib.mlab</td>
<td>1301</td>
</tr>
<tr>
<td>longest_ones()</td>
<td>matplotlib.mlab</td>
<td>1301</td>
</tr>
<tr>
<td>magnitude_spectrum()</td>
<td>matplotlib.mlab</td>
<td>1302</td>
</tr>
<tr>
<td>magnitude_spectrum()</td>
<td>matplotlib.pyplot</td>
<td>1490</td>
</tr>
<tr>
<td>magnitude_spectrum()</td>
<td>matplotlib.axes.Axes</td>
<td>880</td>
</tr>
<tr>
<td>main()</td>
<td>matplotlib.mathtext.Parser</td>
<td>1277</td>
</tr>
<tr>
<td>mainloop()</td>
<td>matplotlib.backend_bases.ShowBase</td>
<td>980</td>
</tr>
<tr>
<td>make_axes()</td>
<td>matplotlib.colorbar</td>
<td>1161</td>
</tr>
<tr>
<td>make_axes_gridspec()</td>
<td>matplotlib.colorbar</td>
<td>1162</td>
</tr>
<tr>
<td>make_compound_path()</td>
<td>matplotlib.path.Path</td>
<td>1369</td>
</tr>
<tr>
<td>make_compound_path_from_polys()</td>
<td>matplotlib.path.Path</td>
<td>1369</td>
</tr>
<tr>
<td>make_image()</td>
<td>matplotlib.image.AxesImage</td>
<td>1237</td>
</tr>
<tr>
<td>make_image()</td>
<td>matplotlib.image.BboxImage</td>
<td>1238</td>
</tr>
<tr>
<td>make_image()</td>
<td>matplotlib.image.FigureImage</td>
<td>1238</td>
</tr>
<tr>
<td>make_image()</td>
<td>matplotlib.image.NonUniformImage</td>
<td>1239</td>
</tr>
<tr>
<td>make_image()</td>
<td>matplotlib.image.PcolorImage</td>
<td>1239</td>
</tr>
<tr>
<td>make_rcparams_key()</td>
<td>matplotlib.font_manager.TempCache</td>
<td>1229</td>
</tr>
<tr>
<td>makeMappingArray()</td>
<td>matplotlib.colors</td>
<td>1172</td>
</tr>
<tr>
<td>margins()</td>
<td>matplotlib.pyplot</td>
<td>1493</td>
</tr>
<tr>
<td>margins()</td>
<td>matplotlib.axes.Axes</td>
<td>883</td>
</tr>
<tr>
<td>margins()</td>
<td>mpl_toolkits.mplot3d.axes3d.Axes3D</td>
<td>561</td>
</tr>
<tr>
<td>markerObject()</td>
<td>matplotlib.backends.backend_pdf.PdfFile</td>
<td>983</td>
</tr>
<tr>
<td>markers()</td>
<td>matplotlib.lines.Line2D</td>
<td>1255</td>
</tr>
<tr>
<td>markers()</td>
<td>matplotlib.markers.MarkerStyle</td>
<td>1263</td>
</tr>
<tr>
<td>MarkerStyle()</td>
<td>matplotlib.markers</td>
<td>1262</td>
</tr>
<tr>
<td>math()</td>
<td>matplotlib.mathtext.Parser</td>
<td>1277</td>
</tr>
<tr>
<td>math_string()</td>
<td>matplotlib.mathtext.Parser</td>
<td>1277</td>
</tr>
<tr>
<td>math_to_image()</td>
<td>matplotlib.mathtext</td>
<td>1280</td>
</tr>
<tr>
<td>MathtextBackend()</td>
<td>matplotlib.mathtext</td>
<td>1273</td>
</tr>
<tr>
<td>MathtextBackendAgg</td>
<td>matplotlib.mathtext</td>
<td>1273</td>
</tr>
<tr>
<td>MathtextBackendBitmap</td>
<td>matplotlib.mathtext</td>
<td>1274</td>
</tr>
<tr>
<td>MathtextBackendCairo</td>
<td>matplotlib.mathtext</td>
<td>1274</td>
</tr>
<tr>
<td>MathtextBackendPath</td>
<td>matplotlib.mathtext</td>
<td>1274</td>
</tr>
<tr>
<td>MathtextBackendPdf</td>
<td>matplotlib.mathtext</td>
<td>1274</td>
</tr>
<tr>
<td>MathtextBackendPs</td>
<td>matplotlib.mathtext</td>
<td>1274</td>
</tr>
<tr>
<td>MathtextBackendSvg</td>
<td>matplotlib.mathtext</td>
<td>1275</td>
</tr>
<tr>
<td>MathTextParser()</td>
<td>matplotlib.mathtext.Parser</td>
<td>1272</td>
</tr>
<tr>
<td>MathTextWarning</td>
<td>matplotlib.mathtext</td>
<td>1272</td>
</tr>
<tr>
<td>matplolib.afm</td>
<td>matplotlib</td>
<td>743</td>
</tr>
<tr>
<td>matplolib.animation</td>
<td>matplotlib</td>
<td>747</td>
</tr>
<tr>
<td>matplolib.artist</td>
<td>matplotlib</td>
<td>757</td>
</tr>
<tr>
<td>matplolib.axis</td>
<td>matplotlib</td>
<td>953</td>
</tr>
<tr>
<td>matplolib.backend_bases</td>
<td>matplotlib</td>
<td>963</td>
</tr>
<tr>
<td>matplolib.backends.backend_pdf</td>
<td>matplotlib</td>
<td>983</td>
</tr>
<tr>
<td>matplolib.backends.backend_qt4agg</td>
<td>matplotlib</td>
<td>981</td>
</tr>
<tr>
<td>matplolib.backends.backend_wxagg</td>
<td>matplotlib</td>
<td>982</td>
</tr>
<tr>
<td>matplolib.cbook</td>
<td>matplotlib</td>
<td>987</td>
</tr>
<tr>
<td>matplolib.cm</td>
<td>matplotlib</td>
<td>1001</td>
</tr>
<tr>
<td>matplolib.collections</td>
<td>matplotlib</td>
<td>1005</td>
</tr>
<tr>
<td>matplolib.colorbar</td>
<td>matplotlib</td>
<td>1159</td>
</tr>
<tr>
<td>matplolib.colors</td>
<td>matplotlib</td>
<td>1165</td>
</tr>
<tr>
<td>matplolib.dates</td>
<td>matplotlib</td>
<td>1175</td>
</tr>
<tr>
<td>matplolib.dviread</td>
<td>matplotlib</td>
<td>1187</td>
</tr>
<tr>
<td>matplolib.figure</td>
<td>matplotlib</td>
<td>1191</td>
</tr>
<tr>
<td>matplolib.finance</td>
<td>matplotlib</td>
<td>1213</td>
</tr>
<tr>
<td>matplolib.font_manager</td>
<td>matplotlib</td>
<td>1225</td>
</tr>
<tr>
<td>matplolib.fontconfig_pattern</td>
<td>matplotlib</td>
<td>1231</td>
</tr>
<tr>
<td>matplolib.gridspec</td>
<td>matplotlib</td>
<td>1233</td>
</tr>
<tr>
<td>matplolib.image</td>
<td>matplotlib</td>
<td>1237</td>
</tr>
<tr>
<td>matplolib.legend</td>
<td>matplotlib</td>
<td>1243</td>
</tr>
<tr>
<td>matplolib.legend_handler</td>
<td>matplotlib</td>
<td>1246</td>
</tr>
<tr>
<td>matplolib.lines</td>
<td>matplotlib</td>
<td>1251</td>
</tr>
<tr>
<td>matplolib.markers</td>
<td>matplotlib</td>
<td>1261</td>
</tr>
<tr>
<td>matplolib.mathtext</td>
<td>matplotlib</td>
<td>1267</td>
</tr>
</tbody>
</table>
mutated() (matplotlib.transforms.Bbox method), 449
mutatedx() (matplotlib.transforms.Bbox method), 449
mutatedy() (matplotlib.transforms.Bbox method), 449
mx2num() (in module matplotlib.dates), 1177

N
Name (class in matplotlib.backends.backend_pdf), 983
name (matplotlib.axes.Axes attribute), 886
name (mpl_toolkits.mplot3d.axes3d.Axes3D attribute), 562, 620
NavigationToolbar2 (class in matplotlib.backend_bases), 974
NavigationToolbar2WxAgg (class in matplotlib.backends.backend_wxagg), 982
NegFil (class in matplotlib.mathtext), 1275
NegFill (class in matplotlib.mathtext), 1275
NegFilll (class in matplotlib.mathtext), 1275
neighbors (matplotlib.tri.Triangulation attribute), 1600
new_axes() (matplotlib.widgets.SpanSelector method), 1624
new_figure_manager() (in module matplotlib.backends.backend_pdf), 985
new_figure_manager() (in module matplotlib.backends.backend_qt4agg), 981
new_figure_manager() (in module matplotlib.backends.backend_wxagg), 982
new_figure_manager_given_figure() (in module matplotlib.backends.backend_pdf), 985
new_figure_manager_given_figure() (in module matplotlib.backends.backend_qt4agg), 981
new_figure_manager_given_figure() (in module matplotlib.backends.backend_wxagg), 982
new_frame_seq() (matplotlib.animation.Animation method), 748
new_frame_seq() (matplotlib.animation.FuncAnimation method), 750
new_gc() (matplotlib.backend_bases.RendererBase method), 979
new_horizontal() (mpl_toolkits.axes_grid.axes_divider.AxesDivider method), 523, 683
new_locator() (mpl_toolkits.axes_grid.axes_divider.Divider method), 521, 681
new_saved_frame_seq() (matplotlib.animation.Animation method), 748
new_saved_frame_seq() (matplotlib.animation.FuncAnimation method), 750
new_subplotspec() (matplotlib.gridspec.GridSpecBase method), 1234
new_timer() (matplotlib.backend_bases.FigureCanvasBase method), 967
new_vertical() (mpl_toolkits.axes_grid.axes_divider.AxesDivider method), 523, 683
no_norm() (in module matplotlib.colors), 1173
Node (class in matplotlib.mathtext), 1275
don_math() (matplotlib.mathtext.Parser method), 1277
NonGuiException, 976
NoNorm (class in matplotlib.colors), 1170
nonsingular() (in module matplotlib.transforms), 463
nonsingular() (matplotlib.dates.AutoDateLocator method), 1180
nonsingular() (matplotlib.dates.DateLocator method), 1179
NonUniformImage (class in matplotlib.image), 1238
norm (matplotlib.cm.ScalarMappable attribute), 1002
norm_angle() (in module mpl_toolkits.mplot3d.art3d), 581, 638
norm_flat() (in module matplotlib.mlab), 1302
norm_text_angle() (in module mpl_toolkits.mplot3d.art3d), 581, 638
Norm (class in matplotlib.patheffects), 1373
Normalize (class in matplotlib.colors), 1170
normalize() (in module matplotlib.colors), 1173
normpdf() (in module matplotlib.mlab), 1302
Null (class in matplotlib.cbook), 989
null() (matplotlib.transforms.Bbox static method), 449
NullFormatter (class in matplotlib.ticker), 1590
NullLocator (class in matplotlib.ticker), 1593
num2date() (in module matplotlib.dates), 1177
num2epoch() (in module matplotlib.dates), 1177

NUM_VERTICES_FOR_CODE (matplotlib.path.Path attribute), 1366

numpy, 2529
numvertices (matplotlib.patches.RegularPolygon attribute), 1360

Index
offset_line() (in module matplotlib.mlab), 1302
OffsetBox (class in matplotlib.offsetbox), 1321
OffsetFrom (class in matplotlib.text), 1576
OffsetImage (class in matplotlib.offsetbox), 1322
OFFSETTEXTPAD (matplotlib.axis.Axis attribute), 953
on_changed() (matplotlib.widgets.Slider method), 1623
on_clicked() (matplotlib.widgets.Button method), 1616
on_clicked() (matplotlib.widgets.CheckButtons method), 1616
on_clicked() (matplotlib.widgets.RadioButton method), 1620
on_mappable_changed() (matplotlib.colorbar.Colorbar method), 1159
on_motion() (matplotlib.offsetbox.DraggableBase method), 1319
on_motion_blit() (matplotlib.offsetbox.DraggableBase method), 1319
on_pick() (matplotlib.offsetbox.DraggableBase method), 1319
on_release() (matplotlib.offsetbox.DraggableBase method), 1319
ontrue() (in module matplotlib.cbook), 996
onHilite() (matplotlib.backend_bases.FigureCanvasBase method), 967
onmove() (matplotlib.widgets.Cursor method), 1617
onmove() (matplotlib.widgets.Lasso method), 1618
onmove() (matplotlib.widgets.LassoSelector method), 1618
onmove() (matplotlib.widgets.MultiCursor method), 1620
onmove() (matplotlib.widgets.RectangleSelector method), 1622
onmove() (matplotlib.widgets.SpanSelector method), 1624
onpick() (matplotlib.lines.VertexSelector method), 1259
onpress() (matplotlib.widgets.LassoSelector method), 1618
onrelease() (matplotlib.widgets.Lasso method), 1618
onrelease() (matplotlib.widgets.LassoSelector method), 1618
onRemove() (matplotlib.backend_bases.FigureCanvasBase method), 967
open_group() (matplotlib.backend_bases.FigureCanvasBase method), 979
Operator (class in matplotlib.backends.backend_pdf), 983
operatorname() (matplotlib.mathtext.Parser method), 1277
option_image_nocomposite() (matplotlib.backend_bases.FigureCanvasBase method), 979
option_scale_image() (matplotlib.backend_bases.FigureCanvasBase method), 979
orientation (matplotlib.patches.RegularPolygon attribute), 1360
OSXInstalledFonts() (in module matplotlib.font_manager), 1229
output_args (matplotlib.animation.FFMpegBase attribute), 749
output_args (matplotlib.animation.ImageMagickBase attribute), 750
output_args (matplotlib.animation.MencoderBase attribute), 751
output_dims (matplotlib.transforms.Transform attribute), 452
over() (in module matplotlib.pyplot), 1494
overlaps() (matplotlib.transforms.BboxBase method), 447
overline() (matplotlib.mathtext.Parser method), 1277
P
p0 (matplotlib.transforms.BboxBase attribute), 447
p1 (matplotlib.transforms.BboxBase attribute), 447
PackerBase (class in matplotlib.offsetbox), 1322
Padded (class in mpl_toolkits.axes_grid.axes_size), 519, 679
padded() (matplotlib.transforms.BboxBase method), 447
PaddedBox (class in matplotlib.offsetbox), 1323
pan() (matplotlib.axis.Axes method), 955
pan() (matplotlib.backend_bases.NavigationToolbar2 method), 975
pan() (matplotlib.ticker.Locator method), 1592
parse() (matplotlib.fontconfig_pattern.FontconfigPatternParser method), 1231
parse() (matplotlib.mathtext.MathTextParser method), 1272
parse() (matplotlib.mathtext.Parser method), 1277
parse_afm() (in module matplotlib.afm), 744
parse_yahoo_historical() (in module matplotlib.finance), 1217
parse_yahoo_historical_ochl() (in module matplotlib.finance), 1217
parse_yahoo_historical_ohlc() (in module matplotlib.finance), 1218
Parser (class in matplotlib.mathtext), 1275
Parser.State (class in matplotlib.mathtext), 1275
parts (matplotlib.type1font.Type1Font attribute), 1611
pass_through (matplotlib.transforms.TransformNode attribute), 445
Patch (class in matplotlib.patches), 1350
Patch3D (class in mpl_toolkits.mplot3d.art3d), 578, 635
Patch3DCollection (class in mpl_toolkits.mplot3d.art3d), 578, 635
patch_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 581, 638
patch_collection_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 581, 639
PatchCollection (class in matplotlib.collections), 1082
PATH, 52, 55, 56, 60
Path (class in matplotlib.path), 1365
Path3DCollection (class in mpl_toolkits.mplot3d.art3d), 579, 636
path_in_path() (in module matplotlib.path), 1372
path_to_3d_segment() (in module matplotlib.path), 1372
path_length() (in module matplotlib.plt), 1303
PathCollection (class in matplotlib.collections), 1092
PathEffectRenderer (class in matplotlib.patheffects), 1373
PathPatch (class in matplotlib.patches), 1354
PathPatch3D (class in mpl_toolkits.mplot3d.art3d), 579, 636
PathPatchEffect (class in matplotlib.patheffects), 1374
paths_to_3d_segments() (in module mpl_toolkits.mplot3d.art3d), 581, 639
pause() (in module matplotlib.pyplot), 1494
PCA (class in matplotlib.mlab), 1288
pchanged() (matplotlib.artist.Artist method), 759
pchanged() (matplotlib.axes.Axes method), 886
pchanged() (matplotlib.collections.AsteriskPolygonCollection method), 1010
pchanged() (matplotlib.collections.BrokenBarHCollection method), 1021
pchanged() (matplotlib.collections.CircleCollection method), 1031
pchanged() (matplotlib.collections.Collection method), 1042
pchanged() (matplotlib.collections.EllipseCollection method), 1053
pchanged() (matplotlib.collections.EventCollection method), 1064
pchanged() (matplotlib.collections.LineCollection method), 1076
pchanged() (matplotlib.collections.PatchCollection method), 1086
pchanged() (matplotlib.collections.PathCollection method), 1097
pchanged() (matplotlib.collections.PolyCollection method), 1107
pchanged() (matplotlib.collections.QuadMesh method), 1118
pchanged() (matplotlib.collections.RegularPolyCollection method), 1129
pchanged() (matplotlib.collections.StarPolygonCollection method), 1140
pchanged() (matplotlib.collections.TriMesh method), 1150
pcolor() (in module matplotlib.image), 1241
pcolormesh() (class in matplotlib.image), 1239
pcolorfast() (matplotlib.axes.Axes method), 889
PcolorImage (class in matplotlib.image), 1239
pcolormesh() (in module matplotlib.pyplot), 890
PDF, 2529
PdfFile (class in matplotlib.backends.backend_pdf), 983
PdfPages (class in matplotlib.backends.backend_pdf), 984
Index 2593
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_day_summary()</td>
<td>matplotlib.finance</td>
<td>1218</td>
</tr>
<tr>
<td>plot_day_summary2()</td>
<td>matplotlib.finance</td>
<td>1219</td>
</tr>
<tr>
<td>plot_day_summary2_ohcl()</td>
<td>matplotlib.finance</td>
<td>1220</td>
</tr>
<tr>
<td>plot_day_summary2_ohlc()</td>
<td>matplotlib.finance</td>
<td>1220</td>
</tr>
<tr>
<td>plot_day_summary_ohcl()</td>
<td>matplotlib.finance</td>
<td>1220</td>
</tr>
<tr>
<td>plot_day_summary_ohlc()</td>
<td>matplotlib.finance</td>
<td>1220</td>
</tr>
<tr>
<td>plot_surface()</td>
<td>mpl_toolkits.mplot3d.Axes3D method</td>
<td>535</td>
</tr>
<tr>
<td>plot_surface()</td>
<td>mpl_toolkits.mplot3d.Axes3D method</td>
<td>535</td>
</tr>
<tr>
<td>plot_trisurf()</td>
<td>mpl_toolkits.mplot3d.Axes3D method</td>
<td>538</td>
</tr>
<tr>
<td>plot_trisurf()</td>
<td>mpl_toolkits.mplot3d.Axes3D method</td>
<td>538</td>
</tr>
<tr>
<td>plot_wireframe()</td>
<td>mpl_toolkits.mplot3d.Axes3D method</td>
<td>534</td>
</tr>
<tr>
<td>plot_wireframe()</td>
<td>mpl_toolkits.mplot3d.Axes3D method</td>
<td>534</td>
</tr>
<tr>
<td>plotfile()</td>
<td>matplotlib.pyplot</td>
<td>1508</td>
</tr>
<tr>
<td>plotting()</td>
<td>matplotlib.pyplot</td>
<td>693</td>
</tr>
<tr>
<td>POINTERS</td>
<td>matplotlib.backend_bases.Cursors attribute</td>
<td>963</td>
</tr>
<tr>
<td>points_in_path()</td>
<td>matplotlib.path</td>
<td>1372</td>
</tr>
<tr>
<td>points_to_pixels()</td>
<td>matplotlib.backend_bases.RendererBase method</td>
<td>979</td>
</tr>
<tr>
<td>points_to_pixels()</td>
<td>matplotlib.patheffects.PathEffectRenderer method</td>
<td>1374</td>
</tr>
<tr>
<td>polar()</td>
<td>matplotlib.pyplot</td>
<td>1509</td>
</tr>
<tr>
<td>PolarAffine</td>
<td>matplotlib.projections.polar</td>
<td>470</td>
</tr>
<tr>
<td>PolarAxes</td>
<td>matplotlib.projections.polar</td>
<td>470</td>
</tr>
<tr>
<td>PolarAxes.InvertedPolarTransform</td>
<td>matplotlib.projections.polar</td>
<td>470</td>
</tr>
<tr>
<td>PolarAxes.PolarAffine</td>
<td>matplotlib.projections.polar</td>
<td>470</td>
</tr>
<tr>
<td>PolarAxes.PolarTransform</td>
<td>matplotlib.projections.polar</td>
<td>470</td>
</tr>
<tr>
<td>PolarAxes.RadialLocator</td>
<td>matplotlib.projections.polar</td>
<td>471</td>
</tr>
<tr>
<td>PolarAxes.ThetaFormatter</td>
<td>matplotlib.projections.polar</td>
<td>472</td>
</tr>
<tr>
<td>PolarTransform</td>
<td>matplotlib.projections.polar</td>
<td>475</td>
</tr>
<tr>
<td>Poly3DCollection</td>
<td>mpl_toolkits.mplot3d.art3d</td>
<td>579</td>
</tr>
<tr>
<td>poly_below()</td>
<td>matplotlib.mlab</td>
<td>1304</td>
</tr>
<tr>
<td>poly_between()</td>
<td>matplotlib.mlab</td>
<td>1304</td>
</tr>
<tr>
<td>poly_collection_2d_to_3d()</td>
<td>matplotlib.mplot3d.art3d</td>
<td>582</td>
</tr>
<tr>
<td>PolarAxes.PolarTransform</td>
<td>matplotlib.projections.polar</td>
<td>475</td>
</tr>
<tr>
<td>PolyCollection</td>
<td>matplotlib.collections</td>
<td>1103</td>
</tr>
<tr>
<td>Polygon</td>
<td>matplotlib.patches</td>
<td>1355</td>
</tr>
<tr>
<td>pop_state()</td>
<td>matplotlib.mathtext.Parser</td>
<td>1277</td>
</tr>
<tr>
<td>popall()</td>
<td>matplotlib.mlab</td>
<td>996</td>
</tr>
<tr>
<td>PowerNorm</td>
<td>matplotlib.colors</td>
<td>1171</td>
</tr>
<tr>
<td>pprint_getters()</td>
<td>matplotlib.artist.ArtistInspector method</td>
<td>763</td>
</tr>
<tr>
<td>pprint_setters()</td>
<td>matplotlib.artist.ArtistInspector method</td>
<td>763</td>
</tr>
<tr>
<td>pprint_setters_rest()</td>
<td>matplotlib.artist.ArtistInspector method</td>
<td>763</td>
</tr>
<tr>
<td>pprint_val()</td>
<td>matplotlib.ticker.LogFormatter method</td>
<td>1591</td>
</tr>
<tr>
<td>pprint_val()</td>
<td>matplotlib.ticker.ScalarFormatter method</td>
<td>1591</td>
</tr>
<tr>
<td>prctile()</td>
<td>matplotlib.mlab</td>
<td>1304</td>
</tr>
<tr>
<td>prctile_rank()</td>
<td>matplotlib.mlab</td>
<td>1304</td>
</tr>
<tr>
<td>prepca()</td>
<td>matplotlib.mlab</td>
<td>1304</td>
</tr>
<tr>
<td>press()</td>
<td>matplotlib.widgets.RectangleSelector method</td>
<td>975</td>
</tr>
<tr>
<td>press()</td>
<td>matplotlib.widgets.SpanSelector method</td>
<td>1622</td>
</tr>
<tr>
<td>press_pan()</td>
<td>matplotlib.backend_bases.NavigationToolbar2 method</td>
<td>975</td>
</tr>
<tr>
<td>press_zoom()</td>
<td>matplotlib.backend_bases.NavigationToolbar2 method</td>
<td>975</td>
</tr>
<tr>
<td>print_cycles()</td>
<td>matplotlib.mlab</td>
<td>996</td>
</tr>
<tr>
<td>print_figure()</td>
<td>matplotlib.mlab</td>
<td>996</td>
</tr>
</tbody>
</table>
Index
Matplotlib, Release 1.4.3

quiverkey() (matplotlib.axes.Axes method), 907
quotes_historical_yahoo() (in module matplotlib.finance), 1221
quotes_historical_yahoo_ochl() (in module matplotlib.finance), 1222
quotes_historical_yahoo_ohlc() (in module matplotlib.finance), 1222

R
RadialLocator (class in matplotlib.projections.polar), 476
RadioButtons (class in matplotlib.widgets), 1620
radius (matplotlib.patches.Circle attribute), 1337
radius (matplotlib.patches.RegularPolygon attribute), 1360
raise_if_exceeds() (matplotlib.ticker.Locator method), 1592
raster graphics, 2529
rc() (in module matplotlib), 739
rc() (in module matplotlib.pyplot), 1515
rc_context (class in matplotlib), 741
rc_context() (in module matplotlib.pyplot), 1516
rc_params() (in module matplotlib), 740
rc_params_from_file() (in module matplotlib.pyplot), 1540
rcdefaults() (in module matplotlib.pyplot), 1516
RcParams (class in matplotlib.pyplot), 740
reParams (in module matplotlib), 739
readonly (matplotlib.path.Path attribute), 1370
rec2csv() (in module matplotlib.mlab), 1306
rec2txt() (in module matplotlib.mlab), 1306
rec_append_fields() (in module matplotlib.mlab), 1307
rec_drop_fields() (in module matplotlib.mlab), 1307
rec_groupby() (in module matplotlib.mlab), 1307
rec_join() (in module matplotlib.mlab), 1307
rec_keep_fields() (in module matplotlib.mlab), 1307
rec_summarize() (in module matplotlib.mlab), 1307
recache() (matplotlib.lines.Line2D method), 1255
recache_always() (matplotlib.lines.Line2D method), 1255
recs_join() (in module matplotlib.mlab), 1307
Rectangle (class in matplotlib.patches), 1357
RectangleSelector (class in matplotlib.widgets), 1620
recursive_remove() (in module matplotlib.cbook), 996
redraw_in_frame() (matplotlib.axes.Axes method), 908
Reference (class in matplotlib.backends.backend_pdf), 985
refine_field() (matplotlib.tri.UniformTriRefiner method), 1605
refine_triangulation() (matplotlib.tri.UniformTriRefiner method), 1606
refresh() (matplotlib.dates.AutoDateLocator method), 1180
refresh() (matplotlib.ticker.Locator method), 1592
register() (matplotlib.animation.MovieWriterRegistry method), 753
register() (matplotlib.mlab.FIFOBu
ner method), 1285
register() (matplotlib.projections.ProjectionRegistry method), 469
register_axis() (matplotlib.spines.Spine method), 1569
register_backend() (in module matplotlib.backends.backend_bases), 981
register_cmap() (in module matplotlib.cm), 1003
register_scale() (in module matplotlib.scale), 468
Registry (class in matplotlib.units), 1614
RegularPolyCollection (class in matplotlib.collections), 1124
RegularPolygon (class in matplotlib.patches), 1359
relativedelta (class in matplotlib.dates), 1184
release() (matplotlib.backend_bases.NavigationToolbar2 method), 975
release() (matplotlib.widgets.LockDraw method), 1619
release() (matplotlib.widgets.RectangleSelector method), 1622
release() (matplotlib.widgets.SpanSelector method), 1624
release_mouse() (matplotlib.backend_bases.FigureCanvasBase method), 968
release_pan() (matplotlib.backend_bases.NavigationToolbar2 method), 976
release_zoom() (matplotlib.backend_bases.NavigationToolbar2 method), 976
relim() (matplotlib.axes.Axes method), 908
reload_library() (in module matplotlib.style), 1571
remove() (matplotlib.artist.Artist method), 759
remove() (matplotlib.axes.Axes method), 908

2596 Index
safe_masked_invalid() (in module matplotlib.cbook), 997
safezip() (in module matplotlib.cbook), 997
Sankey (class in matplotlib.sankey), 1559
save() (matplotlib.animation.Animation method), 748
save_figure() (matplotlib.backend_bases.NavigationToolbar2 method), 976
save_offset() (matplotlib.offsetbox.DraggableAnnotation method), 1318
save_offset() (matplotlib.offsetbox.DraggableBase method), 1319
save_offset() (matplotlib.offsetbox.DraggableOffsetBox method), 1319
savefig() (in module matplotlib.pyplot), 1517
savefig() (matplotlib.backends.backend_pdf.PdfPages method), 984
savefig() (matplotlib.figure.Figure method), 1206
saving() (matplotlib.animation.MovieWriter method), 753
scat() (in module matplotlib.pyplot), 1518
scat() (matplotlib.figure.Figure method), 1207
ScalarFormatter (class in matplotlib.ticker), 1590
ScalarMappable (class in matplotlib.cm), 1001
scale() (matplotlib.transforms.Affine2D method), 457
scale_factors (matplotlib.tri.TriAnalyzer attribute), 1608
scale_factory() (in module matplotlib.scale), 469
ScaleBase (class in matplotlib.scale), 467
Scaled (class in mpl_toolkits.axes_grid.axes_size), 519, 679
scaled() (matplotlib.colors.Normalize method), 1171
ScaledTranslation (class in matplotlib.transforms), 463
scatter() (in module matplotlib.pyplot), 1518
scatter() (matplotlib.axes.Axes method), 909
scatter() (mpl_toolkits.mplot3d.Axes3D method), 533, 593
scatter() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 567, 625
scatter3D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 568, 626
Scheduler (class in matplotlib.cbook), 990
sci() (in module matplotlib.pyplot), 1519
score_family() (matplotlib.font_manager.FontManager method), 1226
score_size() (matplotlib.font_manager.FontManager method), 1226
score_stretch() (matplotlib.font_manager.FontManager method), 1226
score_style() (matplotlib.font_manager.FontManager method), 1226
score_variant() (matplotlib.font_manager.FontManager method), 1226
score_weight() (matplotlib.font_manager.FontManager method), 1226
scotts_factor() (matplotlib.mlab.GaussianKDE method), 1288
scroll_event() (matplotlib.backend_bases.FigureCanvasBase method), 968
SecondLocator (class in matplotlib.dates), 1182
seconds() (in module matplotlib.dates), 1185
segment_hits() (in module matplotlib.lines), 1259
segments_intersect() (in module matplotlib.mlab), 1309
SELECT_REGION (matplotlib.backend_bases.Cursors attribute), 963
semilogx() (in module matplotlib.pyplot), 1520
semilogx() (matplotlib.axes.Axes method), 910
semilogy() (in module matplotlib.pyplot), 1521
semilogy() (matplotlib.axes.Axes method), 912
set() (matplotlib.artist.Artist method), 760
set() (matplotlib.axes.Axes method), 913
set() (matplotlib.collections.AsteriskPolygonCollection method), 1011
set() (matplotlib.collections.BrokenBarHCollection method), 1021
set() (matplotlib.collections.CircleCollection method), 1032
set() (matplotlib.collections.Collection method), 1043
set() (matplotlib.collections.EllipseCollection method), 1053
set() (matplotlib.collections.EventCollection method), 1065
set() (matplotlib.collections.LineCollection method), 1076
set() (matplotlib.collections.PatchCollection method), 1087
set() (matplotlib.collections.PathCollection method), 1097
set() (matplotlib.collections.PolyCollection method), 1108
set() (matplotlib.collections.QuadMesh method), 1119
set() (matplotlib.collections.RegularPolyCollection method), 1130
set() (matplotlib.collections.StarPolygonCollection method), 1141
set() (matplotlib.collections.TriMesh method), 1151
set() (matplotlib.font_manager.TempCache method), 1229
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Line3D method), 577, 635
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Patch3D method), 578, 636
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 578, 636
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Path3DCollection method), 578, 636
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Plate3D method), 579, 636
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Plate3DCollection method), 579, 637
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Plate3D method), 580, 637
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Text3D method), 581, 638
set_aa() (matplotlib.lines.Line2D method), 1255
set_aa() (matplotlib.patches.Patch method), 1353
set_active() (matplotlib.widgets.RectangleSelector method), 1622
set_adjustable() (matplotlib.axes.Axes method), 913
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.artist.Artist method), 760
set_agg_filter() (matplotlib.transforms.Affine2D method), 457
set_agg_filter() (matplotlib.transforms.Bbox method), 449
set_agg_filter() (matplotlib.transforms.TransformWrapper method), 454
set Affero() (matplotlib.collections.AsteriskPolygonCollection method), 1011
set_alpha() (matplotlib.collections.AsteriskPolygonCollection method), 1011
set_alpha() (matplotlib.collections.BrokenBarHCollection method), 1021
set_alpha() (matplotlib.collections.CircleCollection method), 1032
set_alpha() (matplotlib.collections.Collection method), 1043
set_alpha() (matplotlib.collections.EllipseCollection method), 1053
set_alpha() (matplotlib.collections.EventCollection method), 1065
set_alpha() (matplotlib.collections.LineCollection method), 1077
set_alpha() (matplotlib.collections.PatchCollection method), 1087
set_alpha() (matplotlib.collections.PatchCollection method), 1097
set_alpha() (matplotlib.collections.PatchCollection method), 1097
set_alpha() (matplotlib.collections.PatchCollection method), 1097
set_alpha() (matplotlib.collections.PatchCollection method), 1097
set_alpha() (matplotlib.collections.PatchCollection method), 1097
set_alpha() (matplotlib.collections.QuadMesh method), 1108
set_alpha() (matplotlib.collections.RegularPolyCollection method), 1119
set_alpha() (matplotlib.collections.StarPolygonCollection method), 1130
set_alpha() (matplotlib.collections.StarPolygonCollection method), 1141
set_alpha() (matplotlib.collections.TriMesh method), 1151
set_alpha() (matplotlib.collections.BrokenBarHCollection method), 1021
set_alpha() (matplotlib.collections.CircleCollection method), 1032
set_alpha() (matplotlib.collections.EllipseCollection method), 1053
set_alpha() (matplotlib.collections.EventCollection method), 1065
set_antialiased() (matplotlib.collections.PathCollection method), 1087
set_antialiased() (matplotlib.collections.PolyCollection method), 1108
set_antialiased() (matplotlib.collections.QuadMesh method), 1119
set_antialiased() (matplotlib.collections.RegularPolyCollection method), 1130
set_antialiased() (matplotlib.collections.StarPolygonCollection method), 1141
set_antialiased() (matplotlib.collections.TriMesh method), 1151
set_array() (matplotlib.cm.ScalarMappable method), 1002
set_array() (matplotlib.collections.AsteriskPolygonCollection method), 1011
set_array() (matplotlib.collections.BrokenBarHCollection method), 1022
set_array() (matplotlib.collections.CircleCollection method), 1032
set_array() (matplotlib.collections.Collection method), 1043
set_array() (matplotlib.collections.EllipseCollection method), 1054
set_array() (matplotlib.collections.EventCollection method), 1065
set_array() (matplotlib.collections.LineCollection method), 1077
set_array() (matplotlib.collections.PatchCollection method), 1087
set_array() (matplotlib.collections.PathCollection method), 1098
set_array() (matplotlib.collections.QuadMesh method), 1119
set_array() (matplotlib.collections.RegularPolyCollection method), 1130
set_array() (matplotlib.collections.StarPolygonCollection method), 1141
set_array() (matplotlib.collections.TriMesh method), 1151
set_array() (matplotlib.image.FigureImage method), 1238
set_array() (matplotlib.image.NonUniformImage method), 1239
set_array() (matplotlib.image.PcolorImage method), 1240
set_arrowstyle() (matplotlib.patches.FancyArrowPatch method), 1347
set_aspect() (mpl_toolkits.axes_grid.axes_divider.Divider method), 521, 681
set_aspect() (matplotlib.axes.Axes method), 914
set_autoscale_on() (matplotlib.axes.Axes method), 914
set_autoscale_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 568, 626
set_autoscalex_on() (matplotlib.axes.Axes method), 914
set_autoscaley_on() (matplotlib.axes.Axes method), 914
set_autoscalez_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 568, 626
set_axes() (matplotlib.artist.Artist method), 760
set_axes() (matplotlib.axes.Axes method), 915
set_axes() (matplotlib.collections.AsteriskPolygonCollection method), 1011
set_axes() (matplotlib.collections.BrokenBarHCollection method), 1022
set_axes() (matplotlib.collections.CircleCollection method), 1032
set_axes() (matplotlib.collections.Collection method), 1043
set_axes() (matplotlib.collections.EllipseCollection method), 1054
set_axes() (matplotlib.collections.EventCollection method), 1066
set_axes() (matplotlib.collections.LineCollection method), 1077
set_axes() (matplotlib.collections.PatchCollection method), 1087
set_axes() (matplotlib.collections.PathCollection method), 1098
set_axes() (matplotlib.collections.PolyCollection method), 1108
set_axes() (matplotlib.collections.QuadMesh method), 1119
set_axes() (matplotlib.collections.RegularPolyCollection method), 1130
set_axes() (matplotlib.collections.StarPolygonCollection method), 1141
set_axes() (matplotlib.collections.TriMesh method), 1151
set_axes() (matplotlib.lines.Line2D method), 1255
set_axes_locator() (matplotlib.axes.Axes method), 915
set_axis() (matplotlib.dates.AutoDateLocator method), 1180
set_axis() (matplotlib.dates.MillisecondLocator method), 1182
set_axis() (matplotlib.ticker.TickHelper method), 1589
set_axis_bgcolor() (matplotlib.axes.Axes method), 915
set_axis_direction() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 526, 685
set_axis_direction() (mpl_toolkits.axes_grid.axis_artist.AxisLabel method), 527, 687
set_axis_direction() (mpl_toolkits.axes_grid.axis_artist.TickLabels method), 528, 687
set_axis_off() (matplotlib.axes.Axes method), 915
set_axis_off() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 569, 626
set_axison() (matplotlib.axes.Axes method), 915
set_axison() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 569, 626
set_axisbelow() (matplotlib.axes.Axes method), 915
set_axisbelow() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 569, 626
set_axislabel_direction() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 526, 686
set_axislabel_direction() (mpl_toolkits.axes_grid.axis_artist.AxisLabel method), 526, 686
set_axisline_style() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 526, 686
set_bad() (matplotlib.colors.Colormap method), 1167
set_bbox() (matplotlib.text.Text method), 1579
set_bbox_to_anchor() (matplotlib.legend.Legend method), 1246
set_bbox_to_anchor() (matplotlib.offsetbox.AnchoredOffsetbox method), 1316
set_bounds() (matplotlib.patches.FancyBboxPatch method), 1349
set_bounds() (matplotlib.patches.Rectangle method), 1358
set_bounds() (matplotlib.spines.Spine method), 1569
set_bounds() (matplotlib.ticker.TickHelper method), 1589
set_boxstyle() (matplotlib.patches.FancyBboxPatch method), 1167
set_dates() (matplotlib.dates.AutoDateLocator method), 1180
set_dates() (matplotlib.dates.MillisecondLocator method), 1182
set_dates() (matplotlib.ticker.TickHelper method), 1589
set_datestyle() (matplotlib.dates.DayAxisFormatter method), 1201
set_datestyle() (matplotlib.dates.MonthAxisFormatter method), 1203
set_datestyle() (matplotlib.dates.WeekAxisFormatter method), 1209
set_datestyle() (matplotlib.dates.YearAxisFormatter method), 1212
set_datestyle() (matplotlib.dates.Z AxisFormatter method), 1215
set_datestyle() (matplotlib.dates.AttrAxisFormatter method), 1218
set_datestyle() (matplotlib.dates.StrftimeAxisFormatter method), 1220
set_datestyle() (matplotlib.dates.DateFormatter method), 1226
set_datestyle() (matplotlib.dates.YearAxisFormatter method), 1212
set_datestyle() (matplotlib.dates.MonthAxisFormatter method), 1203
set_datestyle() (matplotlib.dates.WeekAxisFormatter method), 1209
set_datestyle() (matplotlib.dates.DayAxisFormatter method), 1201
set_datestyle() (matplotlib.dates.DateFormatter method), 1226
set_datestyle() (matplotlib.dates.YearAxisFormatter method), 1212
set_datestyle() (matplotlib.dates.MonthAxisFormatter method), 1203
set_datestyle() (matplotlib.dates.Z AxisFormatter method), 1215
set_datestyle() (matplotlib.dates.AttrAxisFormatter method), 1218
set_datestyle() (matplotlib.dates.StrftimeAxisFormatter method), 1220
set_c() (matplotlib.lines.Line2D method), 1255
set_canvas() (matplotlib.figure.Figure method), 1207
set_canvas_size() (matplotlib.mathtext.Fonts method), 1270
set_canvas_size() (matplotlib.mathtext.MathtextBackend method), 1273
set_canvas_size() (matplotlib.mathtext.MathtextBackendAgg method), 1274
set_capstyle() (matplotlib.backend_bases.GraphicsContextBase method), 971
set_capstyle() (matplotlib.patches.Patch method), 1353
set_center() (matplotlib.patches.Wedge method), 1362
set_child() (matplotlib.offsetbox.AnchoredOffsetbox method), 1316
set_children() (matplotlib.transforms.TransformNode method), 445
set_clim() (matplotlib.cm.ScalarMappable method), 1002
set_clim() (matplotlib.collections.AsteriskPolygonCollection method), 1011
set_clim() (matplotlib.collections.BrokenBarHCollection method), 1022
set_clim() (matplotlib.collections.CircleCollection method), 1033
set_clim() (matplotlib.collections.Collection method), 1043
set_clim() (matplotlib.collections.EllipseCollection method), 1054
set_clim() (matplotlib.collections.EventCollection method), 1066
set_clim() (matplotlib.collections.LineCollection method), 1077
set_clim() (matplotlib.collections.PatchCollection method), 1088
set_clim() (matplotlib.collections.PathCollection method), 1099
set_clim() (matplotlib.collections.PolyCollection method), 1109
set_clim() (matplotlib.collections.QuadMesh method), 1119
set_clip_box() (matplotlib.artist.Artist method), 760
set_clip_box() (matplotlib.axes.Axes method), 915
set_clip_box() (matplotlib.collections.AsteriskPolygonCollection method), 1011
set_clip_box() (matplotlib.collections.BrokenBarHCollection method), 1022
set_clip_box() (matplotlib.collections.CircleCollection method), 1033
set_clip_box() (matplotlib.collections.Collection method), 1043
set_clip_box() (matplotlib.collections.EllipseCollection method), 1054
set_clip_box() (matplotlib.collections.EventCollection method), 1066
set_clip_box() (matplotlib.collections.LineCollection method), 1077
set_clip_box() (matplotlib.collections.PatchCollection method), 1088
set_clip_box() (matplotlib.collections.PathCollection method), 1099
set_clip_box() (matplotlib.collections.PolyCollection method), 1109
set_clip_box() (matplotlib.collections.QuadMesh method), 1119
set_clip_box() (matplotlib.collections.RegularPolyCollection method), 1130
set_clip_box() (matplotlib.collections.StarPolygonCollection method), 1141
set_clip_box() (matplotlib.collections.TriMesh method), 1152
set_clip_box() (matplotlib.text.Text method), 1580
set_clip_on() (matplotlib.artist.Artist method), 760
set.colorbar() (matplotlib.collections.StarPolygonCollection method), 1142
set.colorbar() (matplotlib.collections.TriMesh method), 1152
set.connectionstyle() (matplotlib.patches.FancyArrowPatch method), 1347
set.contains() (matplotlib.artist.Artist method), 760
set.contains() (matplotlib.axes.Axes method), 915
set.contains() (matplotlib.collections.AsteriskPolygonCollection method), 1012
set.contains() (matplotlib.collections.BrokenBarHCollection method), 1023
set.contains() (matplotlib.collections.CircleCollection method), 1033
set.contains() (matplotlibcollections.Collection method), 1044
set.contains() (matplotlibcollections.EllipseCollection method), 1055
set.contains() (matplotlibcollections.EventCollection method), 1066
set.contains() (matplotlibcollections.LineCollection method), 1078
set.contains() (matplotlibcollections.PatchCollection method), 1088
set.contains() (matplotlibcollections.PathCollection method), 1099
set.contains() (matplotlibcollections.PolyCollection method), 1109
set.contains() (matplotlibcollections.QuadMesh method), 1120
set.contains() (matplotlibcollections.RegularPolygonCollection method), 1131
set.contains() (matplotlibcollections.StarPolygonCollection method), 1142
set.contains() (matplotlibcollections.TriMesh method), 1152
set.cursor() (matplotlib.backend_bases.NavigationToolbar2 method), 976
set.cursor_props() (matplotlib.axes.Axes method), 916
set.dash_capstyle() (matplotlib.lines.Line2D method), 1255
set.dash_joinstyle() (matplotlib.lines.Line2D method), 1255
set.dashdirection() (matplotlib.text.TextWithDash method), 1584
set.dashes() (matplotlib.backend_bases.GraphicsContextBase method), 971
set.dashes() (matplotlib.collections.AsteriskPolygonCollection method), 1012
set.dashes() (matplotlib.collections.BrokenBarHCollection method), 1023
set.dashes() (matplotlib.collections.CircleCollection method), 1034
set.dashes() (matplotlib.collections.Collection method), 1044
set.dashes() (matplotlib.collections.EllipseCollection method), 1055
set.dashes() (matplotlib.collections.EventCollection method), 1067
set.dashes() (matplotlib.collections.LineCollection method), 1078
set.dashes() (matplotlib.collections.PatchCollection method), 1088
set.dashes() (matplotlib.collections.PathCollection method), 1099
set.dashes() (matplotlib.collections.PolyCollection method), 1109
set.dashes() (matplotlib.collections.QuadMesh method), 1110
set.dashes() (matplotlib.collections.RegularPolygonCollection method), 1131
set.dashes() (matplotlib.collections.StarPolygonCollection method), 1142
set.dashes() (matplotlib.collections.TriMesh method), 1153
set.dashlength() (matplotlib.text.TextWithDash method), 1255
set.dashlen() (matplotlib.text.TextWithDash method), 1255
set.dashpad() (matplotlib.text.TextWithDash method), 1584
set.dashpush() (matplotlib.text.TextWithDash method), 1584
set.dashrotation() (matplotlib.text.TextWithDash method), 1584
set_data() (matplotlib.image.FigureImage method), 1238
set_data() (matplotlib.image.NonUniformImage method), 1239
set_data() (matplotlib.image.PcolorImage method), 1240
set_data() (matplotlib.lines.Line2D method), 1255
set_data() (matplotlib.offsetbox.OffsetImage method), 1322
set_data_interval() (matplotlib.axis.Axis method), 955
set_data_interval() (matplotlib.axis.XAxis method), 959
set_data_interval() (matplotlib.axis.YAxis method), 961
set_data_interval() (matplotlib.dates.MicrosecondLocator method), 1182
set_data_interval() (matplotlib.ticker.TickHelper method), 1589
set_default_handler_map() (matplotlib.legend.Legend class method), 1246
set_default_intervals() (matplotlib.axis.Axis method), 956
set_default_intervals() (matplotlib.axis.XAxis method), 959
set_default_intervals() (matplotlib.axis.YAxis method), 961
set_default_locators_and_formatters() (matplotlib.scale.LinearScale method), 467
set_default_locators_and_formatters() (matplotlib.scale.LogScale method), 467
set_default_locators_and_formatters() (matplotlib.scale.ScaleBase method), 468
set_default_locators_and_formatters() (matplotlib.scale.SymmetricalLogScale method), 468
set_default_weight() (matplotlib.font_manager.FontManager method), 1227
set_dpi() (matplotlib.figure.Figure method), 1207
set_dpi_cor() (matplotlib.patches.FancyArrowPatch method), 1347
set_drawstyle() (matplotlib.lines.Line2D method), 1255
set_ec() (matplotlib.patches.Patch method), 1353
set_edgecolor() (matplotlib.collections.AsteriskPolygonCollection method), 1012
set_edgecolor() (matplotlib.collections.BrokenBarHCollection method), 1023
set_edgecolor() (matplotlib.collections.CircleCollection method), 1034
set_edgecolor() (matplotlib.collections.Collection method), 1044
set_edgecolor() (matplotlib.collections.EllipseCollection method), 1055
set_edgecolor() (matplotlib.collections.EventCollection method), 1067
set_edgecolor() (matplotlib.collections.LineCollection method), 1078
set_edgecolor() (matplotlib.collections.PatchCollection method), 1089
set_edgecolor() (matplotlib.collections.PathCollection method), 1099
set_edgecolor() (matplotlib.collections.QuadMesh method), 1110
set_edgecolor() (matplotlib.collections.RegularPolyCollection method), 1131
set_edgecolor() (matplotlib.collections.StarPolygonCollection method), 1142
set_edgecolor() (matplotlib.collections.TriMesh method), 1153
set_edgecolor() (matplotlib.figure.Figure method), 1207
set_edgecolor() (matplotlib.patches.Patch method), 1353
set_edgecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 580, 637
set_edgecolors() (matplotlib.collections.AsteriskPolygonCollection method), 1012
set_edgecolors() (matplotlib.collections.BrokenBarHCollection method), 1023
set_edgecolors() (matplotlib.collections.CircleCollection method), 1034
set_edgecolors() (matplotlib.collections.Collection method), 1045
set_edgecolors() (matplotlib.collections.EllipseCollection method), 1055
set_edgecolors() (matplotlib.collections.EventCollection method), 1067
set_edgecolors() (matplotlib.collections.LineCollection method), 1078
set_edgecolors() (matplotlib.collections.PatchCollection method), 1089
set_edgecolors() (matplotlib.collections.PathCollection method), 1099
set_edgecolors() (matplotlib.collections.PolyCollection method), 1110
set_edgecolors() (matplotlib.collections.QuadMesh method), 1121
set_edgecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 580, 637
set_edgecolor() (mapplotlib.collections.AsteriskPolygonCollection method), 1012
set_edgecolor() (mapplotlib.collections.BrokenBarHCollection method), 1023
set_edgecolor() (mapplotlib.collections.CircleCollection method), 1034
set_edgecolor() (mapplotlib.collections.EllipseCollection method), 1055
set_facecolor() (mapplotlib.collections.EventCollection method), 1067
set_facecolor() (mapplotlib.collections.LineCollection method), 1078
set_facecolor() (mapplotlib.collections.PatchCollection method), 1089
set_facecolor() (mapplotlib.collections.PathCollection method), 1099
set_facecolor() (mapplotlib.collections.PolyCollection method), 1110
set_facecolor() (mapplotlib.collections.QuadMesh method), 1121
set_facecolor() (mapplotlib.collections.RegularPolyCollection method), 1132
set_facecolor() (mapplotlib.collections.StarPolygonCollection method), 1143
set_facecolor() (mapplotlib.collections.TriMesh method), 1153
set_facecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 580, 637
set_facecolors() (mapplotlib.collections.AsteriskPolygonCollection method), 1013
set_facecolors() (mapplotlib.collections.BrokenBarHCollection method), 1023
set_facecolors() (mapplotlib.collections.CircleCollection method), 1034
set_facecolors() (mapplotlib.collections.InsetCollection method), 1045
set_fontsize() (matplotlib.offsetbox.AnnotationBbox method), 1317
set_fontsize() (matplotlib.text.Text method), 1581
set_fontstretch() (matplotlib.text.Text method), 1581
set_fontvariant() (matplotlib.text.Text method), 1581
set_fontweight() (matplotlib.text.Text method), 1581
set_foreground() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_frame_on() (matplotlib.axes.Axes method), 916
set_frame_on() (matplotlib.legend.Legend method), 1246
set_frame_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 569, 626
set_frameon() (matplotlib.figure.Figure method), 1207
set_gamma() (matplotlib.colors.LinearSegmentedColormap method), 1169
set_gid() (matplotlib.artist.Artist method), 761
set_gid() (matplotlib.axes.Axes method), 916
set_gid() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_gid() (matplotlib.collections.AsteriskPolygonCollection method), 1143
set_gid() (matplotlib.collections.TriMesh method), 1153
set_graylevel() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_ha() (matplotlib.text.Text method), 1581
set_hatch() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_hatch() (matplotlib.collections.AsteriskPolygonCollection method), 1013
set_hatch() (matplotlib.collections.BrokenBarHCollection method), 1023
set_hatch() (matplotlib.collections.CircleCollection method), 1034
set_hatch() (matplotlib.collections.Collection method), 1045
set_hatch() (matplotlib.collections.EllipseCollection method), 1056
set_hatch() (matplotlib.collections.EventCollection method), 1067
set_hatch() (matplotlib.collections.LineCollection method), 1079
set_hatch() (matplotlib.collections.PatchCollection method), 1089
set_hatch() (matplotlib.collections.PathCollection method), 1098
set_height() (matplotlib.patches.Patch method), 1100
set_height() (matplotlib.collections.QuadMesh method), 1110
set_height() (matplotlib.collections.RegularPolyCollection method), 1121
set_height() (matplotlib.collections.StarPolygonCollection method), 1132
set_height() (matplotlib.collections.TriMesh method), 1143
degree() (matplotlib.dates.YearBase method), 123
set_history_buttons() (matplotlib.backend_bases.NavigationToolbar2 method), 976
set_horizontal() (mpl_toolkits.axes_grid.axes_divider.Divider method), 521, 681
set_horizontalalignment() (matplotlib.text.Text method), 1581
set_interpolation() (matplotlib.image.NonUniformImage method), 1239
set_joinstyle() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_joinstyle() (matplotlib.patches.Patch method), 1354
set_label() (matplotlib.artist.Artist method), 761
set_label() (matplotlib.axes.Axes method), 916
set_label() (matplotlib.axis.Tick method), 958
set_label() (matplotlib.collections.AsteriskPolygonCollection method), 1013
set_label() (matplotlib.collections.BrokenBarHCollection method), 1024
set_label() (matplotlib.collections.CircleCollection method), 1035
set_label() (matplotlib.collections.Collection method), 1046
set_label() (matplotlib.collections.EllipseCollection method), 1056
set_label() (matplotlib.collections.EventCollection method), 1068
set_label() (matplotlib.collections.LineCollection method), 1079
set_label() (matplotlib.collections.PatchCollection method), 1090
set_label() (matplotlib.collections.PathCollection method), 1100
set_label() (matplotlib.collections.PolyCollection method), 1111
set_label() (matplotlib.collections.QuadMesh method), 1122
set_label() (matplotlib.collections.RegularPolyCollection method), 1132
set_label() (matplotlib.collections.StarPolygonCollection method), 1143
set_label() (matplotlib.collections.TriMesh method), 1154
set_label() (matplotlib.colorbar.ColorbarBase method), 1161
set_label() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 526, 686
set_label1() (matplotlib.axis.Tick method), 958
set_label2() (matplotlib.axis.Tick method), 958
set_label_coords() (matplotlib.axis.Axis method), 956
set_label_position() (matplotlib.axis.XAxis method), 959
set_label_position() (matplotlib.axis.YAxis method), 961
set_label_text() (matplotlib.axis.Axis method), 956
set_linelength() (matplotlib.collections.EventCollection method), 1068
set_lineoffset() (matplotlib.collections.EventCollection method), 1068
set_linespacing() (matplotlib.text.Text method), 1581
set_linestyle() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_linestyle() (matplotlib.collections.AsteriskPolygonCollection method), 1013
set_linestyle() (matplotlib.collections.BrokenBarHCollection method), 1024
set_linestyle() (matplotlib.collections.CircleCollection method), 1035
set_linestyle() (matplotlib.collections.EventCollection method), 1068
set_linestyle() (matplotlib.collections.LineCollection method), 1079
set_linestyle() (matplotlib.collections.PatchCollection method), 1090
set_linestyle() (matplotlib.collections.PathCollection method), 1100
set_linestyle() (matplotlib.collections.PolyCollection method), 1111
set_linestyle() (matplotlib.collections.QuadMesh method), 1122
set_linestyle() (matplotlib.collections.RegularPolyCollection method), 1132
set_linestyle() (matplotlib.collections.StarPolygonCollection method), 1143
set_linestyle() (matplotlib.collections.TriMesh method), 1154
set_linestyle() (matplotlib.colorbar.ColorbarBase method), 1161
set_linestyle() (matplotlib.collections.PolyCollection method), 1111
set_linestyle() (matplotlib.collections.QuadMesh method), 1122
set_linestyle() (matplotlib.collections.RegularPolyCollection method), 1132
set_linestyle() (matplotlib.collections.StarPolygonCollection method), 1143
set_linestyle() (matplotlib.collections.TriMesh method), 1154
set_linewidth() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_linewidth() (matplotlib.collections.AsteriskPolygonCollection method), 1014
set_linewidth() (matplotlib.collections.BrokenBarHCollection method), 1024
set_linewidth() (matplotlib.collections.CircleCollection method), 1035
set_linewidth() (matplotlib.collections.EllipseCollection method), 1056
set_linewidth() (matplotlib.collections.EventCollection method), 1068
set_linewidth() (matplotlib.collections.LineCollection method), 1079
set_linewidth() (matplotlib.collections.PatchCollection method), 1090
set_linewidth() (matplotlib.collections.PolyCollection method), 1111
set_linewidth() (matplotlib.collections.QuadMesh method), 1122
set_linewidth() (matplotlib.collections.RegularPolyCollection method), 1133
set_linewidth() (matplotlib.collections.StarPolygonCollection method), 1144
set_linewidth() (matplotlib.collections.TriMesh method), 1154
set_linewidth() (matplotlib.collections.TriMesh method), 1154
set_linewidth() (matplotlib.lines.Line2D method), 1256
set_linewidth() (matplotlib.patches.Patch method), 1354
set_linewidths() (matplotlib.collections.AsteriskPolygonCollection method), 1014
set_linewidths() (matplotlib.collections.BrokenBarHCollection method), 1024
set_linewidths() (matplotlib.collections.CircleCollection method), 1035
set_linewidths() (matplotlib.collections.Collection method), 1046
set_linewidths() (matplotlib.collections.EllipseCollection method), 1056
set_linewidths() (matplotlib.collections.EventCollection method), 1068
set_linewidths() (matplotlib.collections.LineCollection method), 1080
set_linewidths() (matplotlib.collections.PatchCollection method), 1090
set_linewidths() (matplotlib.collections.PathCollection method), 1100
set_linewidths() (matplotlib.collections.PolyCollection method), 1111
set_linewidths() (matplotlib.collections.QuadMesh method), 1122
set_linewidths() (matplotlib.collections.RegularPolyCollection method), 1133
set_linewidths() (matplotlib.collections.StarPolygonCollection method), 1144
set_linewidths() (matplotlib.collections.TriMesh method), 1154
set_lw() (matplotlib.collections.BrokenBarHCollection method), 1014
set_lw() (matplotlib.collections.CircleCollection method), 1024
set_lw() (matplotlib.collections.EllipseCollection method), 1035
set_lw() (matplotlib.collections.EventCollection method), 1046
set_lw() (matplotlib.collections.LineCollection method), 1057
set_lw() (matplotlib.collections.PatchCollection method), 1068
set_lw() (matplotlib.collections.PolyCollection method), 1144
set_lw() (matplotlib.collections.QuadMesh method), 1154
set_lw() (matplotlib.collections.RegularPolyCollection method), 1161
set_lw() (matplotlib.collections.StarPolygonCollection method), 1172
set_lw() (matplotlib.collections.TriMesh method), 1183
set_lw() (matplotlib.collections.PatchCollection method), 1080
set_lw() (matplotlib.collections.PathCollection method), 1090
set_lw() (matplotlib.collections.PolyCollection method), 1101
set_lw() (matplotlib.collections.QuadMesh method), 1122
set_lw() (matplotlib.collections.RegularPolyCollection method), 1133
set_lw() (matplotlib.collections.StarPolygonCollection method), 1144
set_lw() (matplotlib.collections.TriMesh method), 1154
set_lw() (matplotlib.lines.Line2D method), 1256
set_lw() (matplotlib.patches.Patch method), 1354
set_major_formatter() (matplotlib.axis.Axis method), 956
set_major_locator() (matplotlib.axis.Axis method), 956
set_marker() (matplotlib.lines.Line2D method), 1256
set_markeredgecolor() (matplotlib.lines.Line2D method), 1256
set_markeredgewidth() (matplotlib.lines.Line2D method), 1257
set_markerfacecolor() (matplotlib.lines.Line2D method), 1257
set_markerfacecoloralt() (matplotlib.lines.Line2D method), 1257
set_markersize() (matplotlib.lines.Line2D method), 1257
set_markevery() (matplotlib.lines.Line2D method), 1257
set_mask() (matplotlib.tri.Triangulation method), 1600
set_matrix() (matplotlib.transforms.Affine2D method), 457
set_mec() (matplotlib.lines.Line2D method), 1258
set_message() (matplotlib.backend_bases.NavigationToolbar2 method), 976
set_mew() (matplotlib.lines.Line2D method), 1258
set_mfc() (matplotlib.lines.Line2D method), 1258
set_mfcalt() (matplotlib.lines.Line2D method), 1258
set_minimumdescent() (matplotlib.offsetbox.TextArea method), 1324
set_minor_formatter() (matplotlib.axis.Axis method), 956
set_minor_locator() (matplotlib.axis.Axis method), 956
set_mutation_aspect() (matplotlib.patches.FancyArrowPatch method), 1347
set_mutation_aspect() (matplotlib.patches.FancyBboxPatch method), 1350
set_mutation_scale() (matplotlib.patches.FancyArrowPatch method), 1347
set_mutation_scale() (matplotlib.patches.FancyBboxPatch method), 1350
set_name() (matplotlib.font_manager.FontProperties method), 1229
set_name() (matplotlib.text.Text method), 1581
set_navigate() (matplotlib.axes.Axes method), 916
set_navigate_mode() (matplotlib.axes.Axes method), 916
set_norm() (matplotlib.cm.ScalarMappable method), 1002
set_norm() (matplotlib.collections.AsteriskPolygonCollection method), 1014
set_norm() (matplotlib.collections.BrokenBarHCollection method), 1024
set_norm() (matplotlib.collections.CircleCollection method), 1035
set_norm() (matplotlib.collections.Collection method), 1046
set_norm() (matplotlib.collections.EllipseCollection method), 1057
set_norm() (matplotlib.collections.EventCollection method), 1069
set_norm() (matplotlib.collections.LineCollection method), 1080
set_offsets() (matplotlib.collections.TriMesh method), 1154
set_orientation() (matplotlib.collections.EventCollection method), 1069
set_over() (matplotlib.colors.Colormap method), 1167
set_pad() (matplotlib.axis.Tick method), 958
set_pad() (mpl_toolkits.axes_grid.axis_artist.AxisLabel method), 527, 687
setPane_color() (mpl_toolkits.mplot3d.axis3d.Axis method), 576, 634
setPane_pos() (mpl_toolkits.mplot3d.axis3d.Axis method), 576, 634
set_params() (matplotlib.ticker.MaxNLocator method), 1594
set_path_circle() (matplotlib.spines.Spine method), 1569
set_path_line() (matplotlib.spines.Spine method), 1569
set_patchA() (matplotlib.patches.FancyArrowPatch method), 1347
set_patchB() (matplotlib.patches.FancyArrowPatch method), 1347
set_path_effects() (matplotlib.artist.Artist method), 761
set_path_effects() (matplotlib.axes.Axes method), 917
set_path_effects() (matplotlib.collections.AsteriskPolygonCollection method), 1014
set_path_effects() (matplotlib.collections.BrokenBarHCollection method), 1025
set_path_effects() (matplotlib.collections.CircleCollection method), 1035
set_path_effects() (matplotlib.collections.Collection method), 1046
set_path_effects() (matplotlibcollections.AsteriskPolygonCollection method), 1057
set_path_effects() (matplotlibcollections.BrokenBarHCollection method), 1069
set_path_effects() (matplotlibcollections.CircleCollection method), 1080
set_path_effects() (matplotlibcollections. PatchCollection method), 1090
set_path_effects() (matplotlibcollections.PolyCollection method), 1101
set_path_effects() (matplotlibcollections.QuadMesh method), 1112
set_path_effects() (matplotlibcollections.RegularPolygonCollection method), 1133
set_path_effects() (matplotlibcollections.StarPolygonCollection method), 1144
set_paths() (matplotlibcollections.AsteriskPolygonCollection method), 1014
set_paths() (matplotlibcollections.BrokenBarHCollection method), 1025
set_paths() (matplotlibcollections.CircleCollection method), 1036
set_paths() (matplotlibcollections.Collection method), 1046
set_paths() (matplotlibcollections.EllipseCollection method), 1057
set_paths() (matplotlibcollections.LineCollection method), 1069
set_paths() (matplotlibcollections.PatchCollection method), 1080
set_paths() (matplotlibcollections.PathCollection method), 1090
set_paths() (matplotlibcollections.PolyCollection method), 1101
set_paths() (matplotlibcollections.QuadMesh method), 1112
set_paths() (matplotlibcollections.RegularPolygonCollection method), 1133
set_paths() (matplotlibcollections.StarPolygonCollection method), 1144
set_paths() (matplotlibcollections.TriMesh method), 1155
set_picker() (matplotlib.artist.Artist method), 761
<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_picker() (matplotlib.axes.Axes method)</td>
<td>917</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.AsteriskPolygonCollection method)</td>
<td>1014</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.BrokenBarHCollection method)</td>
<td>1025</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.CircleCollection method)</td>
<td>1036</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.Collection method)</td>
<td>1046</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.EllipseCollection method)</td>
<td>1057</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.EventCollection method)</td>
<td>1069</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.LineCollection method)</td>
<td>1080</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.PatchCollection method)</td>
<td>1091</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.PathCollection method)</td>
<td>1101</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.PolyCollection method)</td>
<td>1112</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.QuadMesh method)</td>
<td>1122</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.RegularPolyCollection method)</td>
<td>1133</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.StarPolygonCollection method)</td>
<td>1144</td>
</tr>
<tr>
<td>set_picker() (matplotlib.collections.TriMesh method)</td>
<td>1155</td>
</tr>
<tr>
<td>set_picker() (matplotlib.lines.Line2D method)</td>
<td>1258</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.AsteriskPolygonCollection method)</td>
<td>1081</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.BrokenBarHCollection method)</td>
<td>1091</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.CircleCollection method)</td>
<td>1101</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.Collection method)</td>
<td>1112</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.EllipseCollection method)</td>
<td>1112</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.EventCollection method)</td>
<td>1122</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.LineCollection method)</td>
<td>1133</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.PatchCollection method)</td>
<td>1144</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.PathCollection method)</td>
<td>1155</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.collections.PolyCollection method)</td>
<td>1169</td>
</tr>
<tr>
<td>set_pickradius() (matplotlib.Patches.Bbox method)</td>
<td>449</td>
</tr>
<tr>
<td>set_position() (matplotlib.axes.Axes method)</td>
<td>917</td>
</tr>
<tr>
<td>set_position() (matplotlib.patches.Patch method)</td>
<td>1569</td>
</tr>
<tr>
<td>set_position() (mpl_toolkits.axes_grid.axes_divider.Divider method)</td>
<td>521, 682</td>
</tr>
<tr>
<td>set_powerlimits() (matplotlib.ticker.ScalarFormatter method)</td>
<td>1591</td>
</tr>
<tr>
<td>set_radius() (matplotlib.Patches.Circle method)</td>
<td>1337</td>
</tr>
<tr>
<td>set_radius() (matplotlib.Patches.Wedge method)</td>
<td>1362</td>
</tr>
<tr>
<td>set_rasterization_zorder() (matplotlib.axes.Axes method)</td>
<td>917</td>
</tr>
</tbody>
</table>
set_rasterized() (matplotlib.artist.Artist method), 761
set_rasterized() (matplotlib.axes.Axes method), 917
set_rasterized() (matplotlib.collections.AsteriskPolygonCollection method), 1015
set_rasterized() (matplotlib.collections.BrokenBarHCollection method), 1025
set_rasterized() (matplotlib.collections.CircleCollection method), 1036
set_rasterized() (matplotlib.collections.Collection method), 1047
set_rasterized() (matplotlib.collections.EllipseCollection method), 1057
set_rasterized() (matplotlib.collections.EventCollection method), 1069
set_rasterized() (matplotlib.collections.LineCollection method), 1081
set_rasterized() (matplotlib.collections.PathCollection method), 1091
set_rasterized() (matplotlib.collections.PatchCollection method), 1101
set_rasterized() (matplotlib.collections.PolyCollection method), 1112
set_rasterized() (matplotlib.collections.QuadMesh method), 1123
set_rasterized() (matplotlib.collections.RegularPolyCollection method), 1134
set_rasterized() (matplotlib.collections.StarPolygonCollection method), 1145
set_rasterized() (matplotlib.collections.TriMesh method), 1155
set_rgrids() (matplotlib.projections.polar.PolarAxes method), 472
set_rlabel_position() (matplotlib.projections.polar.PolarAxes method), 473
set_rotate_label() (mpl_toolkits.mplot3d.axis3d.Axis method), 576, 634
set_rotation() (matplotlib.text.Text method), 1581
set_rotation_mode() (matplotlib.text.Text method), 1581
set_scale() (matplotlib.axis.Axis method), 956
set科學() (matplotlib.ticker.ScalarFormatter method), 1591
set_segments() (matplotlib.collections.EventCollection method), 1070
set_segments() (matplotlib.collections.LineCollection method), 1081
set_segments() (mpl_toolkits.mplot3d.art3d.Line3DCollection method), 578, 635
set_size() (matplotlib.font_manager.FontProperties method), 1229
set_size() (matplotlib.axis.Axis method), 1208
set_size() (matplotlib.text.Text method), 1582
set_size_inches() (matplotlib.figure.Figure method), 1208
set_sizes() (matplotlib.collections.AsteriskPolygonCollection method), 1015
set_sizes() (matplotlib.collections.BrokenBarHCollection method), 1025
set_sizes() (matplotlib.collections.CircleCollection method), 1036
set_sizes() (matplotlib.collections.PatchCollection method), 1102
set_sizes() (matplotlib.collections.PolyCollection method), 1112
set_sizes() (matplotlib.collections.RegularPolyCollection method), 1134
set_sizes() (matplotlib.collections.StarPolygonCollection method), 1145
set_sketch_params() (matplotlib.artist.Artist method), 762
set_sketch_params() (mpl_toolkits.mplot3d.axis3d.Axis method), 918
set_sketch_params() (matplotlib.backend_bases.GraphicsContextBase method), 972
set_sketch_params() (matplotlib.collections.AsteriskPolygonCollection method), 1015
set_sketch_params() (matplotlib.collections.BrokenBarHCollection method), 1026
set_sketch_params() (matplotlib.collections.LineCollection method), 1081
set_snap() (matplotlib.figure.Figure method), 1208
set_style() (matplotlib.style.Style method), 1219
set_subplots_adjust() (matplotlib.figure.Figure method), 1214
set_subplots_adjust() (matplotlib.pyplot.SubplotParamDict method), 1221
set_title() (matplotlib.text.Text method), 1582
set_title_position() (matplotlib.text.Text method), 1583
set_title_weight() (matplotlib.font_manager.FontProperties method), 1208
set_transform() (matplotlib.font_manager.FontProperties method), 1229
set_ticker() (matplotlib.axis.Axis method), 1252
set_tight_layout() (matplotlib.figure.Figure method), 1214
set_tries() (matplotlib.pyplot.SubplotParamDict method), 1221
set_url() (matplotlib.text.Text method), 1582
set_urltext() (matplotlib.text.Text method), 1582
set_urltext_position() (matplotlib.text.Text method), 1582
set_urltext_weight() (matplotlib.font_manager.FontProperties method), 1208
set_yscale() (matplotlib.axis.Axis method), 1262
set_xlabel() (matplotlib.axis.Axis method), 1262
set_ylabel() (matplotlib.axis.Axis method), 1262
set_xlim() (matplotlib.axis.Axis method), 1262
set_xlim() (matplotlib.figure.Figure method), 1262
set_xlim() (matplotlib.pyplot.SubplotParamDict method), 1262
set_ylim() (matplotlib.axis.Axis method), 1262
set_ylim() (matplotlib.figure.Figure method), 1262
set_ylim() (matplotlib.pyplot.SubplotParamDict method), 1262
set_xmargin() (matplotlib.axis.Axis method), 1271
set_xmargin() (matplotlib.figure.Figure method), 1271
set_xmargin() (matplotlib.pyplot.SubplotParamDict method), 1271
set_xscale() (matplotlib.axis.Axis method), 1262
set_xticks() (matplotlib.axis.Axis method), 1279
set_xticks() (matplotlib.figure.Figure method), 1279
set_xticks() (matplotlib.pyplot.SubplotParamDict method), 1279
set_yticks() (matplotlib.axis.Axis method), 1279
set_yticks() (matplotlib.figure.Figure method), 1279
set_yticks() (matplotlib.pyplot.SubplotParamDict method), 1279
set_ymargin() (matplotlib.axis.Axis method), 1271
set_ymargin() (matplotlib.figure.Figure method), 1271
set_ymargin() (matplotlib.pyplot.SubplotParamDict method), 1271
set_theta1() (matplotlib.patches.Wedge method), 1362
set_theta2() (matplotlib.patches.Wedge method), 1362
set_theta_direction() (matplotlib.projections.polar.PolarAxes method), 474
set_theta_offset() (matplotlib.projections.polar.PolarAxes method), 474
set_theta_zero_location() (matplotlib.projections.polar.PolarAxes method), 474
set_thetagrids() (matplotlib.projections.polar.PolarAxes method), 474
set_tick_out() (mpl_toolkits.axes_grid.axis_artist.Ticks method), 527, 686
set_tick_params() (matplotlib.axis.Axis method), 956
set_ticklabel_direction() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 526, 686
set_ticklabels() (matplotlib.axis.Axis method), 956
set_ticks() (matplotlib.axis.Axis method), 957
set_ticks() (matplotlib.colorbar.ColorbarBase method), 1161
set_ticks_position() (matplotlib.axis.XAxis method), 959
set_ticks_position() (matplotlib.axis.YAxis method), 961
set_ticks_size() (mpl_toolkits.axes_grid.axis_artist.Ticks method), 527, 686
set_tight_layout() (matplotlib.figure.Figure method), 1208
set_title() (matplotlib.axes.Axes method), 918
set_title() (matplotlib.legend.Legend method), 1246
set_title() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 569, 627
set_top_view() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 569, 627
set_transform() (matplotlib.artist.Artist method), 762
set_transform() (matplotlib.axes.Axes method), 918
set_transform() (matplotlib.collections.AsteriskPolygonCollection method), 1015
set_transform() (matplotlib.collections.BrokenBarHCollection method), 1026
set_transform() (matplotlib.collections.CircleCollection method), 1037
set_transform() (matplotlib.collections.Collection method), 1047
set_transform() (matplotlib.collections.EllipseCollection method), 1058
set_transform() (matplotlib.collections.EventCollection method), 1070
set_transform() (matplotlib.collections.LineCollection method), 1081
set_transform() (matplotlib.collections.PatchCollection method), 1092
set_transform() (matplotlib.collections.PathCollection method), 1102
set_transform() (matplotlib.collections.PolyCollection method), 1113
set_transform() (matplotlib.collections.QuadMesh method), 1123
set_transform() (matplotlib.collections.RegularPolyCollection method), 1134
set_transform() (matplotlib.collections.StarPolygonCollection method), 1145
set_transform() (matplotlib.collections.TriMesh method), 1156
set_transform() (matplotlib.lines.Line2D method), 1258
set_transform() (matplotlib.offsetbox.AuxTransformBox method), 1318
set_transform() (matplotlib.offsetbox.DrawingArea method), 1320
set_transform() (matplotlib.offsetbox.TextArea method), 1324
set_transform() (matplotlib.text.TextWithDash method), 1585
set_tzinfo() (matplotlib.dates DateFormatter method), 1178
set_tzinfo() (matplotlib.dates DateLocator method), 1179
set_under() (matplotlib.colors Colormap method), 1167
set_unit() (matplotlib.text OffsetFrom method), 1576
set_units() (matplotlib.axis Axis method), 957
set_url() (matplotlib.artist Artist method), 762
set_url() (matplotlib.axes Axes method), 918
set_url() (matplotlib.backends backend_bases GraphicsContextBase method), 972
set_url() (matplotlib.collections AsteriskPolygonCollection method), 1015
set_url() (matplotlib.collections BrokenBarHCollection method), 1026
set_url() (matplotlib.collections CircleCollection method), 1037
set_url() (matplotlib.collections Collection method), 1048
set_url() (matplotlib.collections EllipseCollection method), 1058
set_url() (matplotlib.collections EventCollection method), 1070
set_url() (matplotlib.collections LineCollection method), 1081
set_url() (matplotlib.collections PatchCollection method), 1092
set_url() (matplotlib.collections PathCollection method), 1102
set_url() (matplotlib.collections PolyCollection method), 1113
set_url() (matplotlib.collections QuadMesh method), 1124
set_url() (matplotlib.collections RegularPolyCollection method), 1135
set_url() (matplotlib.collections StarPolygonCollection method), 1146
set_url() (matplotlib.collections TriMesh method), 1156
set_url() (matplotlib.collections Patches method), 1167
set_url() (matplotlib.collections Polygon method), 1178
set_url() (matplotlib.collections PolyCollection method), 1189
set_url() (matplotlib.collections QuadMesh method), 1190
set_url() (matplotlib.collections RegularPolyCollection method), 1201
set_url() (matplotlib.collections StarPolygonCollection method), 1212
set_url() (matplotlib.collections TriMesh method), 1223
set_url() (mpl_toolkits.mplot3d art3d Poly3DCollection method), 580, 638
set_url() (mpl_toolkits.mplot3d art3d Poly3DCollection method), 1026
set_url() (mpl_toolkits.mplot3d art3d Poly3DCollection method), 1026
set_url() (mpl_toolkits.mplot3d art3d Poly3DCollection method), 1026
set_url() (mpl_toolkits.mplot3d art3d Poly3DCollection method), 1026
set_url() (mpl_toolkits.mplot3d art3d Poly3DCollection method), 1026
set_urls() (matplotlib.collections EllipseCollection method), 1058
set_urls() (matplotlib.collections EventCollection method), 1070
set_urls() (matplotlib.collections LineCollection method), 1081
set_urls() (matplotlib.collections PatchCollection method), 1092
set_urls() (matplotlib.collections PathCollection method), 1102
set_urls() (matplotlib.collections PolyCollection method), 1113
set_urls() (matplotlib.collections QuadMesh method), 1124
set_urls() (matplotlib.collections RegularPolyCollection method), 1135
set_urls() (matplotlib.collections StarPolygonCollection method), 1146
set_urls() (matplotlib.collections TriMesh method), 1156
set_url() (matplotlib.text Text method), 1582
set_urls() (matplotlib.widgets Slider method), 1623
set_variant() (matplotlib.ticker ScalarFormatter method), 1229
set_variant() (matplotlib.text.Text method), 1582
set_vertical() (mpl_toolkits.axes_grid axes_divider Divider method), 522, 682
set_vertical() (mpl_toolkits.axes_grid axes_divider Divider method), 1582
setVerticalAlignment() (matplotlib.text Text method), 1582
set_view_interval() (matplotlib.axis Axis method), 957
set_view_interval() (matplotlib.axis XAxis method), 960
set_view_interval() (matplotlib.axis.YAxis method), 961
set_view_interval() (matplotlib.dates.MicrosecondLocator method), 1182
set_view_interval() (matplotlib.ticker.TickHelper method), 1589
set_view_interval() (mpl_toolkits.mplot3d.axis3d.Axis method), 577, 634
set_visible() (matplotlib.artist.Artist method), 762
set_visible() (matplotlib.axes.Axes method), 919
set_visible() (matplotlib.collections.AsteriskPolygonCollection method), 1016
set_visible() (matplotlib.collections.BrokenBarHCollection method), 1026
set_visible() (matplotlib.collections.CircleCollection method), 1037
set_visible() (matplotlib.collections.Collection method), 1048
set_visible() (matplotlib.collections.EllipseCollection method), 1058
set_visible() (matplotlib.collections.EventCollection method), 1070
set_visible() (matplotlib.collections.LineCollection method), 1081
set_visible() (matplotlib.collections.PatchCollection method), 1092
set_visible() (matplotlib.collections.PathCollection method), 1102
set_visible() (matplotlib.collections.PolyCollection method), 1113
set_visible() (matplotlib.collections.QuadMesh method), 1124
set_visible() (matplotlib.collections.RegularPolyCollection method), 1135
set_visible() (matplotlib.collections.StarPolygonCollection method), 1146
set_visible() (matplotlib.collections.TriMesh method), 1156
set_weight() (matplotlib.font_manager.FontProperties method), 1229
set_weight() (matplotlib.text.Text method), 1582
set_width() (matplotlib.offsetbox.OffsetBox method), 1321
set_width() (matplotlib.patches.FancyBboxPatch method), 1350
set_width() (matplotlib.patches.Patch method), 1359
set_width() (matplotlib.patches.Wedge method), 1362
set_width_ratios() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1234
set_window_title() (matplotlib.backend_bases.FigureCanvasBase method), 968
set_window_title() (matplotlib.backend_bases.FigureManagerBase method), 969
set_x() (matplotlib.patches.FancyBboxPatch method), 1350
set_x() (matplotlib.patches.Rectangle method), 1359
set_x() (matplotlib.text.Text method), 1582
set_x() (matplotlib.text.TextWithDash method), 1585
set_xbound() (matplotlib.axes.Axes method), 919
set_xdata() (matplotlib.lines.Line2D method), 1258
set_xlabel() (matplotlib.axes.Axes method), 919
set_xlim() (matplotlib.axes.Axes method), 919
set_xlim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 570, 627
set_xlim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 570, 628
set_xmargin() (matplotlib.axes.Axes method), 920
set_xscale() (matplotlib.axes.Axes method), 920
set_xscale() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 570, 627
set_xticklabels() (matplotlib.axes.Axes method), 920
set_xticks() (matplotlib.axes.Axes method), 922
set_xy() (matplotlib.patches.Polygon method), 1357
set_xy() (matplotlib.patches.Rectangle method), 1359
set_xy() (matplotlib.patches.FancyBboxPatch method), 1350
set_y() (matplotlib.patches.Rectangle method), 1359
set_y() (matplotlib.text.Text method), 1582
set_y() (matplotlib.text.TextWithDash method), 1585
set_ybound() (matplotlib.axes.Axes method), 922
set_ydata() (matplotlib.lines.Line2D method), 1258
set_ylabel() (matplotlib.axes.Axes method), 922
set_ylim() (matplotlib.axes.Axes method), 922
set_ylim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 570, 628
set_ylim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 570, 628
Index
simple_linear_interpolation() (in module matplotlib.cbook), 997
SimpleLineShadow (class in matplotlib.path.effects), 1374
SimplePatchShadow (class in matplotlib.path.effects), 1374
simplify_threshold (matplotlib.path.Path attribute), 1370
single_shot (matplotlib.backend_bases.TimerBase attribute), 980
size (matplotlib.dviread.DviFont attribute), 1188
size (matplotlib.transforms.BboxBase attribute), 447
skew() (matplotlib.transforms.Affine2D method), 458
skew_deg() (matplotlib.transforms.Affine2D method), 458
Slider (class in matplotlib.widgets), 1622
slopes() (in module matplotlib.mlab), 1309
sort() (matplotlib.cbook.Sorter method), 990
Sorter (class in matplotlib.cbook), 990
soundex() (in module matplotlib.cbook), 997
space() (matplotlib.mathtext.Parser method), 1277
span_where() (matplotlib.collections.BrokenBarHCollection static method), 1026
SpanSelector (class in matplotlib.widgets), 1623
specgram() (in module matplotlib.mlab), 1309
specgram() (matplotlib.axes.Axes method), 925
spectral() (in module matplotlib.mlab), 1526
Spine (class in matplotlib.spines), 1567
splitx() (matplotlib.transforms.BboxBase method), 448
splity() (matplotlib.transforms.BboxBase method), 448
spring() (in module matplotlib.pyplot), 1526
spy() (in module matplotlib.pyplot), 1526
spy() (matplotlib.axes.Axes method), 927
sqrt() (matplotlib.mathtext.Parser method), 1277
SsGlue (class in matplotlib.mathtext), 1277
Stack (class in matplotlib.cbook), 990
stackplot() (in module matplotlib.pyplot), 1527
stackplot() (matplotlib.axes.Axes method), 928
stackrel() (matplotlib.mathtext.Parser method), 1277
StandardPsFonts (class in matplotlib.mathtext), 1278
StarPolygonCollection (class in matplotlib.collections), 1135
start() (matplotlib.backend_bases.TimerBase method), 981
start_event_loop() (matplotlib.backend_bases.FigureCanvasBase method), 968
start_event_loop_default() (matplotlib.backend_bases.FigureCanvasBase method), 968
start_filter() (matplotlib.backend_bases.RendererBase method), 979
start_group() (matplotlib.mathtext.Parser method), 1277
start_pan() (matplotlib.axes.Axes method), 929
start_rasterizing() (matplotlib.backend_bases.RendererBase method), 979
stem() (in module matplotlib.pyplot), 1528
stem() (matplotlib.axes.Axes method), 929
step (matplotlib.backend_bases.MouseEvent attribute), 974
step() (in module matplotlib.pyplot), 1529
step() (matplotlib.axes.Axes method), 930
stineman_interpl() (in module matplotlib.mlab), 1311
StixFonts (class in matplotlib.mathtext), 1278
StixSansFonts (class in matplotlib.mathtext), 1279
STOP (matplotlib.path.Path attribute), 1366
stop() (matplotlib.backend_bases.TimerBase method), 981
stop() (matplotlib.cbook.Scheduler method), 990
stop_event_loop() (matplotlib.backend_bases.FigureCanvasBase method), 968
stop_event_loop_default() (matplotlib.backend_bases.FigureCanvasBase method), 968
stop_filter() (matplotlib.backend_bases.RendererBase method), 979
stop_rasterizing() (matplotlib.backend_bases.RendererBase method), 979
Stream (class in matplotlib.backends.backend_pdf), 985
streamplot() (in module matplotlib.pyplot), 1529
streamplot() (matplotlib.axes.Axes method), 930
strftime() (matplotlib.dates.DateFormatter method), 1178
stride_repeat() (in module matplotlib.mlab), 1311
stride_windows() (in module matplotlib.mlab), 1312
string_width_height() (matplotlib.afm.AFM method), 744
strip_math() (in module matplotlib.cbook), 997
strip_math() (matplotlib.backend_bases.RendererBase method), 979
Stroke (class in matplotlib.path.effects), 1375
subplot() (in module matplotlib.pyplot), 1530
subplot2grid() (in module matplotlib.pyplot), 1532
subplot_tool() (in module matplotlib.pyplot), 1532
SubplotDivider (class in mpl_toolkits.axes_grid.axes_divider), 522, 682
SubplotParams (class in matplotlib.figure), 1210
subplots() (in module matplotlib.pyplot), 1532
subplots_adjust() (in module matplotlib.pyplot), 1534
subplots_adjust() (matplotlib.figure.Figure method), 1208
SubplotSpec (class in matplotlib.gridspec), 1234
SubplotTool (class in matplotlib.widgets), 1624
subs() (matplotlib.ticker.LogLocator method), 1594
subsuper() (matplotlib.mathtext.Parser method), 1277
SubSuperCluster (class in matplotlib.mathtext), 1279
summer() (in module matplotlib.pyplot), 1534
supported_formats (matplotlib.animation.FFMpegFileWriter attribute), 749
supported_formats (matplotlib.animation.ImageMagickFileWriter attribute), 750
supported_formats (matplotlib.animation.MencoderFileWriter attribute), 751
supports_blit (matplotlib.backend_bases.FigureCanvasBase attribute), 969
suptitle() (in module matplotlib.pyplot), 1534
suptitle() (matplotlib.figure.Figure method), 1208
SVG, 2529
switch_backend() (in module matplotlib.pyplot), 1534
switch_backends() (matplotlib.backend_bases.FigureCanvasBase method), 969
switch_orientation() (matplotlib.collections.EventCollection method), 1070
symbol() (matplotlib.mathtext.Parser method), 1277
SymLogNorm (class in matplotlib.colors), 1171
SymmetricalLogScale (class in matplotlib.scale), 468
T
table() (in module matplotlib.pyplot), 1535
table() (matplotlib.axes.Axes method), 931
target (matplotlib.mathtext.BakomaFonts attribute), 1268
TempCache (class in matplotlib.font_manager), 1229
test_lines_dists() (in module mpl_toolkits.mplot3d.proj3d), 583, 640
test_proj() (in module mpl_toolkits.mplot3d.proj3d), 583, 640
test_proj_draw_axes() (in module mpl_toolkits.mplot3d.proj3d), 583, 640
test_proj_make_M() (in module mpl_toolkits.mplot3d.proj3d), 583, 640
test_rot() (in module mpl_toolkits.mplot3d.proj3d), 583, 640
test_world() (in module mpl_toolkits.mplot3d.proj3d), 583, 640
texname (matplotlib.dviread.DviFont attribute), 1187, 1188
Text (class in matplotlib.text), 1576
text() (in module matplotlib.pyplot), 1535
text() (matplotlib.axes.Axes method), 932
text() (matplotlib.figure.Figure method), 1209
text() (mpl_toolkits.mplot3d.Axes3D method), 551, 611
text() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 573, 630
text2D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 573, 630
Text3D (class in mpl_toolkits.mplot3d.art3d), 580, 638
text3D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 574, 631
text_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 582, 639
TextArea (class in matplotlib.offsetbox), 1323
TextWithDash (class in matplotlib.text), 1583
Tfm (class in matplotlib.dviread), 1189
ThetaFormatter (class in matplotlib.projections.polar), 476
thetagrids() (in module matplotlib.pyplot), 1536
thumbnail() (in module matplotlib.image), 1241
Tick (class in matplotlib.axis), 957
tick_bottom() (matplotlib.axis.XAxis method), 960
tick_left() (matplotlib.axis.YAxis method), 961
tick_params() (in module matplotlib.pyplot), 1537
tick_params() (matplotlib.axes.Axes method), 933
tick_params() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 574, 631
tick_right() (matplotlib.axis.YAxis method), 961
tick_top() (matplotlib.axis.XAxis method), 960
tick_update_position() (in module
mpl_toolkits.mplot3d.axis3d), 577, 635
tick_values() (matplotlib.ticker.AutoMinorLocator
method), 1595
tick_values() (matplotlib.ticker.FixedLocator
method), 1593
tick_values() (matplotlib.ticker.IndexLocator
method), 1593
tick_values() (matplotlib.ticker.LinearLocator
method), 1593
tick_values() (matplotlib.ticker.Locator method),
1592
tick_values() (matplotlib.ticker.LogLocator
method), 1594
tick_values() (matplotlib.ticker.MaxNLocator
method), 1594
tick_values() (matplotlib.ticker.MultipleLocator
method), 1594
tick_values() (matplotlib.ticker.NullLocator
method), 1593
Ticker (class in matplotlib.axis), 959
TickHelper (class in matplotlib.ticker), 1589
ticklabel_format() (in module matplotlib.pyplot),
1537
ticklabel_format() (matplotlib.axes.Axes method),
933
ticklabel_format() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 574, 631
TickLabels (class in mpl_toolkits.axes_grid.axis_artist),
527, 687
Ticks (class in mpl_toolkits.axes_grid.axis_artist),
527, 686
TIFF, 2530
tight_layout() (in module matplotlib.pyplot), 1538
tight_layout() (matplotlib.figure.Figure method),
1210
tight_layout() (matplotlib.gridspec.GridSpec
method), 1233
TimedAnimation (class in matplotlib.animation),
753
Timeout (class in matplotlib.cbook), 991
TimerBase (class in matplotlib.backend_bases), 980
title() (in module matplotlib.pyplot), 1538
Tk, 2530
to_filehandle() (in module matplotlib.cbook), 997
to_mask() (matplotlib.mathtext.MathTextParser
method), 1272
to_png() (matplotlib.mathtext.MathTextParser
method), 1272
to_polygons() (matplotlib.path.Path method), 1370
to_rgb() (matplotlib.colors.ColorConverter method),
1166
to_rgba() (matplotlib.cm ScalarMappable method),
1002
to_rgba() (matplotlib.collections.AsteriskPolygonCollection
method), 1016
to_rgba() (matplotlib.collections.BrokenBarHCollection
method), 1027
to_rgba() (matplotlib.collections.CircleCollection
method), 1037
to_rgba() (matplotlib.collections.Collection
method), 1048
to_rgba() (matplotlib.collections.EllipseCollection
method), 1058
to_rgba() (matplotlib.collections.EventCollection
method), 1070
to_rgba() (matplotlib.collections.LineCollection
method), 1082
to_rgba() (matplotlib.collections.PathCollection
method), 1092
to_rgba() (matplotlib.collections.PatchCollection
method), 1102
to_rgba() (matplotlib.collections.PolyCollection
method), 1113
to_rgba() (matplotlib.collections.QuadMesh
method), 1124
to_rgba() (matplotlib.collections.RegularPolyCollection
method), 1135
to_rgba() (matplotlib.collections.StarPolygonCollection
method), 1146
to_rgba() (matplotlib.collections.TriMesh method),
1156
to_rgba() (matplotlib.colors.ColorConverter
method), 1167

Index 2627
to_rgba() (matplotlib.mathtext.MathTextParser method), 1272
to_rgba_array() (matplotlib.colors.ColorConverter method), 1167
to_values() (matplotlib.transforms.AffineTransform method), 456
todate (class in matplotlib.cbook), 997
todatetime (class in matplotlib.cbook), 997
tofloat (class in matplotlib.cbook), 997
toggle() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 526, 686
toint (class in matplotlib.cbook), 997
tostr (class in matplotlib.cbook), 997
tostr() (matplotlib.mlab.FormatFormatStr method), 1286
tostr() (matplotlib.mlab.FormatInt method), 1286
tostr() (matplotlib.mlab.FormatObj method), 1287
tostr() (matplotlib.mlab.FormatString method), 1287
toval() (matplotlib.mlab.FormatBool method), 1287
toval() (matplotlib.mlab.FormatDate method), 1287
toval() (matplotlib.mlab.FormatFloat method), 1287
toval() (matplotlib.mlab.FormatInt method), 1287
toval() (matplotlib.mlab.FormatObj method), 1287
Transform (class in matplotlib.transforms), 451
transform() (in module mpl_toolkits.mplot3d.proj3d), 583, 640
transform() (matplotlib.transforms.AffineTransform method), 454
transform() (matplotlib.transforms.IdentityTransform method), 458
transform() (matplotlib.transforms.Transform method), 452
transform() (matplotlib.type1font.Type1Font method), 1611
transform_affine() (matplotlib.transforms.AffineTransform method), 456
transform_affine() (matplotlib.transforms.AffineTransform method), 454
transform_affine() (matplotlib.transforms.CompositeGenericTransform method), 461
transform_affine() (matplotlib.transforms.IdentityTransform method), 459
transform_path() (matplotlib.transforms.AffineTransform method), 455
transform_path() (matplotlib.transforms.AffineTransform method), 459
transform_path() (matplotlib.transforms.Transform method), 453
transform_path_affine() (matplotlib.transforms.AffineTransform method), 455
transform_path_affine() (matplotlib.transforms.AffineTransform method), 459
transform_path_affine() (matplotlib.transforms.IdentityTransform method), 459
Index

Matplotlib, Release 1.4.3

Type1Font (class in matplotlib.type1font), 1611
Unichr_safe() (in module matplotlib.mathtext), 1280
Unicode_safe() (in module matplotlib.cbook), 998
UnicodeFonts (class in matplotlib.mathtext), 1279
UniformTriRefiner (class in matplotlib.tri), 1605
union() (matplotlib.transforms.BboxBase static method), 448
unique() (in module matplotlib.cbook), 998
unit() (matplotlib.transforms.Bbox static method), 449
unit_bbox() (in module mpl_toolkits.mplot3d.axes3d), 576, 633
unit_circle() (matplotlib.path.Path class method), 1370
unit_circle_righthalf() (matplotlib.path.Path class method), 1371
unit_cube() (mpl_toolkits.mplot3d.axes3dAxes3D method), 575, 633
unit_rectangle() (matplotlib.path.Path class method), 1371
unit_regular_asterisk() (matplotlib.path.Path class method), 1371
unit_regular_polygon() (matplotlib.path.Path class method), 1371
unit_regular_star() (matplotlib.path.Path class method), 1371
unknown_symbol() (matplotlib.mathtext.Parser method), 1277
unmasked_index_ranges() (in module matplotlib.cbook), 998
update() (matplotlib.artist.Artist method), 762
update() (matplotlib.axes.Axes method), 946
update() (matplotlib.backend_bases.NavigationToolbar2 method), 976
update() (matplotlib.collections.AsteriskPolygonCollection method), 1016
update() (matplotlib.collections.BrokenBarHCollection method), 1027
update() (matplotlib.collections.CircleCollection method), 1037
update() (matplotlib.collections.Collection method), 1048
update() (matplotlib.collections.EllipseCollection method), 1059
update() (matplotlib.collections.EventCollection method), 1071
update() (matplotlib.collections.LineCollection method), 1082
update() (matplotlib.collections.PatchCollection method), 1092
update() (matplotlib.collections.PathCollection method), 1103
update() (matplotlib.collections.PolyCollection method), 1114
update() (matplotlib.collections.QuadMesh method), 1124
update() (matplotlib.collections.RegularPolyCollection method), 1135
update() (matplotlib.collections.StarPolygonCollection method), 1146
update() (matplotlib.collections.TriMesh method), 1156
update() (matplotlib.figure.SubplotParams method), 1211
update() (matplotlib.colorbar.Colorbar method), 1233
update_background() (matplotlib.widgets.RectangleSelector method), 1622
update() (matplotlib.widgets.SpanSelector method), 1624
update_background() (matplotlib.widgets.LassoSelector method), 1619
update_background() (matplotlib.widgets.RectangleSelector method), 1622
update_background() (matplotlib.widgets.SpanSelector method), 1624
update_bbox_position_size() (matplotlib.text.Annotation method), 1576
update_bbox_position_size() (matplotlib.text.Text method), 1583
update_bruteform() (matplotlib.colorbar.Colorbar method), 1159
update_coords() (matplotlib.text.TextWithDash method), 1585
update_datalim() (matplotlib.axes.Axes method), 947
update_datalim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 575, 633
update_datalim_bounds() (matplotlib.axes.Axes method), 947
update_datalim_numerix() (matplotlib.axes.Axes method), 947
matplotlib.collections.AsteriskPolygonCollection.use_cmex (matplotlib.mathtext.UnicodeFonts attribute), 1280

update_scalarmappable() (matplotlib.collections.BrokenBarHCollection method), 1027

update_scalarmappable() (matplotlib.collections.CircleCollection method), 1038

update_scalarmappable() (matplotlib.collections.EventCollection method), 1071

update_scalarmappable() (matplotlib.collections.LineCollection method), 1082

update_scalarmappable() (matplotlib.collections.PatchCollection method), 1092

update_scalarmappable() (matplotlib.collections.PathCollection method), 1103

update_scalarmappable() (matplotlib.collections.PolyCollection method), 1114

update_scalarmappable() (matplotlib.collections.QuadMesh method), 1124

update_scalarmappable() (matplotlib.collections.RegularPolyCollection method), 1135

update_scalarmappable() (matplotlib.collections.StarPolygonCollection method), 1146

update_scalarmappable() (matplotlib.collections.TriMesh method), 1156

update_ticks() (matplotlib.colorbar.ColorbarBase method), 1161

update_units() (matplotlib.axis.Axis method), 957

violin() (matplotlib.axes.Axes method), 947

violin_stats() (in module matplotlib.cbook), 998

view_init() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 575, 633

view_limits() (matplotlib.ticker.LinearLocator method), 1593

view_limits() (matplotlib.ticker.LogLocator method), 1592

view_limits() (matplotlib.ticker.MaxNLocator method), 1594

view_limits() (matplotlib.ticker.MultipleLocator method), 1595

view_transformation() (in module mpl_toolkits.mplot3d.proj3d), 583, 640

viewlim_to_dt() (matplotlib.dates.DateLocator method), 1179

vlist (class in matplotlib.mathtext), 1280

V

validCap (matplotlib.lines.Line2D attribute), 1259

validCap (matplotlib.patches.Patch attribute), 1354

validJoin (matplotlib.patches.Patch attribute), 1259

validJoin (matplotlib.patches.Patch attribute), 1354

value_escape() (in module matplotlib.fontconfig_pattern), 1231

decoder (in module matplotlib.fontconfig_pattern), 1231

Vbox (class in matplotlib.mathtext), 1280

Vcentered (class in matplotlib.mathtext), 1280

decoder (in module matplotlib.fontconfig_pattern), 1231

vcentered (class in matplotlib.mathtext), 1280

vec_pad_ones() (in module mpl_toolkits.mplot3d.proj3d), 583, 640

vector graphics, 2530

vector_lengths() (in module matplotlib.mlab), 1312

VertexSelector (class in matplotlib.patches), 1259

vertices (matplotlib.path.Path attribute), 1371

Vf (class in matplotlib.dviread), 1189

view_init() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 575, 633

view_limits() (matplotlib.ticker.LinearLocator method), 1593

view_limits() (matplotlib.ticker.LogLocator method), 1592

view_limits() (matplotlib.ticker.MaxNLocator method), 1594

view_limits() (matplotlib.ticker.MultipleLocator method), 1595

view_transformation() (in module mpl_toolkits.mplot3d.proj3d), 583, 640

viewlim_to_dt() (matplotlib.dates.DateLocator method), 1179

violin() (matplotlib.axes.Axes method), 947

violin_stats() (in module matplotlib.cbook), 998

violinplot() (in module matplotlib.pyplot), 1550

violinplot() (matplotlib.axes.Axes method), 948

vlines() (in module matplotlib.pyplot), 1550

vlines() (matplotlib.axes.Axes method), 949

Vlist (class in matplotlib.mathtext), 1280
vlist_out() (matplotlib.mathtext.Ship method), 1278
volume_overlay() (in module matplotlib.finance),
1223
volume_overlay2() (in module matplotlib.finance),
1223
volume_overlay3() (in module matplotlib.finance),
1224
vpack() (matplotlib.mathtext.Vlist method), 1280
VPacker (class in matplotlib.offsetbox), 1324
Vrule (class in matplotlib.mathtext), 1280

W
waitforbuttonpress() (in module matplotlib.pyplot),
1552
waitforbuttonpress() (matplotlib.figure.Figure method), 1210
waittime (matplotlib.cbook.Idle attribute), 989
warn_deprecated() (in module matplotlib.cbook),
999
Wedge (class in matplotlib.patches), 1361
wedge() (matplotlib.path.Path class method), 1371
WeekdayLocator (class in matplotlib.dates), 1181
weeks() (in module matplotlib.dates), 1185
weight_as_number() (in module matplotlib.font_manager),
1230
Widget (class in matplotlib.widgets), 1625
width (matplotlib.dviread.Tfm attribute), 1189
width (matplotlib.transforms.BboxBase attribute),
448
widths (matplotlib.dviread.DviFont attribute), 1188
win32FontDirectory() (in module matplotlib.font_manager),
1230
win32InstalledFonts() (in module matplotlib.font_manager),
1231
window_hanning() (in module matplotlib.mlab),
1312
window_none() (in module matplotlib.mlab), 1312
winter() (in module matplotlib.pyplot), 1553
withSimplePatchShadow (class in matplotlib.pathEffects),
1375
withStroke (class in matplotlib.pathEffects), 1376
world_transformation() (in module mpl_toolkits.mplot3d.proj3d),
583, 640
wrap() (in module matplotlib.cbook), 1000
write() (matplotlib.backends.backend_pdf.Stream method),
985
write_png() (matplotlib.image.FigureImage method), 1238
writeInfoDict() (matplotlib.backends.backend_pdf.PdfFile method), 983
writeTrailer() (matplotlib.backends.backend_pdf.PdfFile method), 983
writeXref() (matplotlib.backends.backend_pdf.PdfFile method), 983
wxpython, 2530
wxWidgets, 2530

X
x (matplotlib.backend_bases.LocationEvent attribute), 973
x (matplotlib.backend_bases.MouseEvent attribute), 974
x0 (matplotlib.transforms.BboxBase attribute), 448
x1 (matplotlib.transforms.BboxBase attribute), 448
XAxis (class in matplotlib.axis), 959
XAxis (class in mpl_toolkits.mplot3d.axis3d), 577, 634
xaxis_date() (matplotlib.axes.Axes method), 950
xaxis_inverted() (matplotlib.axes.Axes method), 950
xcorr() (in module matplotlib.pyplot), 1553
xcorr() (matplotlib.axes.Axes method), 950
xdata (matplotlib.backend_bases.LocationEvent attribute), 973
xdata (matplotlib.backend_bases.MouseEvent attribute), 974
xkcd() (in module matplotlib.pyplot), 1553
xlabel() (in module matplotlib.pyplot), 1554
xlat() (matplotlib.cbook.Xlator method), 991
Xlator (class in matplotlib.cbook), 991
xlim() (in module matplotlib.pyplot), 1554
xmax (matplotlib.transforms.BboxBase attribute),
448
xmin (matplotlib.transforms.BboxBase attribute), 448
xscale() (in module matplotlib.pyplot), 1554
XTick (class in matplotlib.axis), 960
xticks() (in module matplotlib.pyplot), 1555
xy (matplotlib.patches.Polygon attribute), 1357
xy (matplotlib.patches.Rectangle attribute), 1359
xy (matplotlib.patches.RegularPolygon attribute),
1360
xy() (matplotlib.cbook.MemoryMonitor method),
989
xyann (matplotlib.offsetbox.AnnotationBbox attribute), 1317
xyann (matplotlib.text.Annotation attribute), 1576

Y
y (matplotlib.backend_bases.LocationEvent attribute), 973
y (matplotlib.backend_bases.MouseEvent attribute), 974
y0 (matplotlib.transforms.BboxBase attribute), 448
y1 (matplotlib.transforms.BboxBase attribute), 448
YAArrow (class in matplotlib.patches), 1363
YAxis (class in matplotlib.axis), 960
YAxis (class in mpl_toolkits.mplot3d.axis3d), 577, 634
yaxis_date() (matplotlib.axes.Axes method), 951
yaxis_inverted() (matplotlib.axes.Axes method), 951
ydata (matplotlib.backend_bases.LocationEvent attribute), 973
ydata (matplotlib.backend_bases.MouseEvent attribute), 974
YearLocator (class in matplotlib.dates), 1180
ylabel() (in module matplotlib.pyplot), 1555
ylim() (in module matplotlib.pyplot), 1556
ymax (matplotlib.transforms.BboxBase attribute), 448
ymin (matplotlib.transforms.BboxBase attribute), 448
yscale() (in module matplotlib.pyplot), 1556
YTick (class in matplotlib.axis), 961
yticks() (in module matplotlib.pyplot), 1557

Z
zalpha() (in module mpl_toolkits.mplot3d.art3d), 582, 639
ZAxis (class in mpl_toolkits.mplot3d.axis3d), 577, 634
zaxis_date() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 575, 633
zaxis_inverted() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 575, 633
zoom() (matplotlib.axis.Axes method), 957
zoom() (matplotlib.backend_bases.NavigationToolbar2 method), 976
zoom() (matplotlib.ticker.Locator method), 1593
zorder (matplotlib.collections.AsteriskPolygonCollection attribute), 1016
zorder (matplotlib.collections.BrokenBarHCollection attribute), 1027
zorder (matplotlib.collections.CircleCollection attribute), 1038
zorder (matplotlib.collections.Collection attribute), 1048
zorder (matplotlib.collections.EllipseCollection attribute), 1059
zorder (matplotlib.collections.EventCollection attribute), 1071
zorder (matplotlib.collections.LineCollection attribute), 1082
zorder (matplotlib.collections.PathCollection attribute), 1092
zorder (matplotlib.collections.PatchCollection attribute), 1103
zorder (matplotlib.collections.PolyCollection attribute), 1114
zorder (matplotlib.collections.QuadMesh attribute), 1124
zorder (matplotlib.collections.RegularPolyCollection attribute), 1135
zorder (matplotlib.collections.StarPolygonCollection attribute), 1146
zorder (matplotlib.collections.TriMesh attribute), 1156
zorder (matplotlib.image.FigureImage attribute), 1238
zorder (matplotlib.legend.Legend attribute), 1246
zorder (matplotlib.lines.Line2D attribute), 1259
zorder (matplotlib.offsetbox.AnchoredOffsetbox attribute), 1316
zorder (matplotlib.offsetbox.AnnotationBbox attribute), 1317
zorder (matplotlib.patches.Patch attribute), 1354
zorder (matplotlib.text.Text attribute), 1583
ZORDER (mpl_toolkits.axis_grid.axis_artist.AxisArtist attribute), 525, 685