More about Homework #4

Professor Hugh C. Lauer

CS-1004 — Introduction to Programming for Non-Majors

(Slides include materials from *Python Programming: An Introduction to Computer Science*, 2nd edition, by John Zelle and copyright notes by Prof. George Heineman of Worcester Polytechnic Institute)

More Notes about Program Design

- No prescriptive rule about how to write a program
- Every problem is different
- Every programmer thinks about a problem differently
- Best we can do is offer *guidelines* about how to get started, how to go about it.

Guidelines from Chapter 2

- **§2.1 – p. 28**
- **Analyze the Problem**
  - *Exactly* what problem are you trying to solve
  - *In detail*
- **Specifications**
  - What your program will *do*
  - Not *how* it does it, but *what* it accomplishes
- **Create Design**
  - For your program
  - *How* your program works
- **Test and Debug**
  - Nobody gets it right the first time
- **Maintain the program**
  - Programs usually evolve and adapt

Other ways to think about problem

- What kind of data is needed to get desired result?
- Working backwards …
- … how to get that data together
- … and then work backwards from there
- … how to get *that* data together
- … Etc.

"Top-down" design

- **See §9.3**
- **Partition into sub-problems**
  - "Separation of concerns"
- **Attack each sub-problem separately**
  - With own test cases
- **Progressively work down to finer detail**
  - Get each part working before moving on
- **Pull it all back together!**

Homework #4

- **What was required output?**
  - A: Graphs of #s of games versus #s of throws
- **What function can produce that output?**
  - A: **`pyplot.plot()`**
- **What arguments does `pyplot.plot()` need to produce that output?**
  - A: Lists
  - Specifically, y-value list contains numbers of games indexed by numbers of throws
  - What about x-value list?
    - Is it needed at all? If not, why? If so, how organized?
  - List value indicates number of games ending in that number of throws
    - Separate lists for wins and losses
- **Now we are in position to start to design program!**
  - Working backwards from end result!

Homework #4

Easy parts:–

- **Playing one game**
  - Multiple rolls of dice
  - Returning a pair of results
- **Rolling the dice**
  - Two independent dice
    - Values 1 – 6 each
  - Use **`random.randint()`**
    - Variant of **`random.randrange()`**
    - See Python docs §9.6

Harder parts:–

- **Setting up the lists**
- **Growing lists to accommodate long games**

Questions?

Review of Functions

- **Reading assignment — Chapter 6**
  - §6.1 – §6.3 are easy & mostly review
  - §6.4 – §6.6:– read carefully

- **What is a *function*?**
  - A sequence of one or more lines of Python code that can be invoked as a group under a single name (from Week 1)
  - A subprogram to accomplish a specific, defined "sub-computation"
  - (Usually) takes parameters
  - (Usually) returns a result

- **What is a *method*?**
  - A function that lives inside of an object
  - Has access to the "innards" of *that specific* object

Why do we …

- **… have functions?**
  - To avoid writing the same code in multiple places
  - To encapsulate a sub-idea of the program or problem
  - To solve a subset of a problem and put it out of our minds while we concentrate on the rest of the job!
  - To make it easier to build up a working program …
  - … to solve interesting problems …
  - … … that are too difficult to solve "by hand"

- **… have parameters?**
  - To be able to supply values and objects to function
  - So we can apply function to broader range of things

- **… have results?**
  - To pass answers back from functions!

Scope

- **Definition:– the region where an identifier is meaningful**
  - I.e., where it can be used to refer to the same value, object, function, etc.

- **Scope rule #1:–**
  - A variable name (or any other identifier) defined *inside* a function cannot be seen *outside* of a function
  - I.e., it is *local* to that function
    - See bottom p. 175
  - Variable name *ceases to exist* when function returns!
  - Comes back into existence as a *brand new* variable on next invocation
  - This rule is common to (essentially) all modern programming languages

- **Reminder:– Definition of *variable***
  - *Name* used to refer to a value, object, list, function, etc.

Why do we have Scope Rule #1?

- **If two people working on different functions in the same project, …**
- **… don't want the *local* identifiers of one to interfere with the other**
- **Even one person working on a large project needs to be able to put details of a function out of mind while working on other functions**
  - Include names of all of its local variables

Scope (continued)

- **Scope Rule #2:–**

- **A function can see the *names* (i.e., variables, objects, functions, etc.) defined *outside* that function but in same module**
  - Even if defined later in the module
  - But not inside another function!

- **Why Scope Rule #2?**
  - So functions can store values (and objects) that persist from one call to next

Importing and Scope

- **What does import do?**

- **Answer:–**
  - Makes names of one module available to another
  - I.e., adds to the "scope" of the module

`import scopeExample`

- Makes identifier "**`scopeExample`**" available to current scope
- Implicitly makes all components of **`scopeExample`** also available
  - **`scopeExample.m`**
  - **`scopeExample.appendSin()`**
  - **`scopeExample.printM()`**

`import scopeExample as SE`

- Same as **`import ScopeExample`** but gives it a shorthand name '**SE**'

Importing and Scope (continued)

```
from scopeExample import appendSin
from scopeExample import printM
```

- Imports identifiers `appendSin()`, `printM()`
  - Adds to current scope
- … but *NOT* `scopeExample`

Good/bad Python practice

- **The *Python* experts advise AGAINST USING**
  `from moduleName import *`
- **Why?**
  - Because it will bring a large number of unknown variables/names into your scope
  - If you (unwittingly) re-use any name …
  - … could cause truly mysterious havoc to your program
- **But we all do it all the time:–**
  `from graphics import *`
- **It is a good idea to avoid it as much as possible!**

Important convention

- **In *Python*, it is best practice to use '_' at start of function or variable to name to mean**
  - This name should be treated as private!
  - Don't use it!
  - Don't mess with it!
- **Many other programming languages have means to enforce privacy**
  - *Python* does not!
  - It is just a matter of good practice

Questions?

Today

- Reminder:– Line limit on functions
- Mutable and Immutable
- Tuples

Line limits on function

- **See p. 190**
- **… "not to put too fine a point on it — this function is just too long"**
- **What is wrong with functions that are too long?**
  - Discussion
- **What do you do if a function evolves to become too long?**
  - Discussion
- **Note penalty in HW5**
  - Does not include comments
  - 25 lines maximum per function

Today

- Line limit on functions
- **Mutable and Immutable**
- **Tuples**

"Mutable" *vs.* "Immutable"

- **Fundamental to *Python***
- **Only glancing reference in textbook**
- **Definition:– *Variable***

  Symbolic name used to refer to a value!
- **More precisely:– *Variable***

  Symbolic name used to refer to a value or an object!

What kinds of values in *Python*?

- **ints — i.e., the integers**
  - Arbitrary precision
- **floats — i.e., floating point numbers in IEEE 754 standard**
  - http://en.wikipedia.org/wiki/IEEE_floating_point
- **bools — i.e., `True` and `False`**
- **Strings — Any sequence of (printable) characters**
  - String literal — enclosed in quote marks
  - String itself — not including the enclosing quotes
- **Tuples**
  - See below!
- **All of these are *Immutable***
  - I.e., they never change
  - Assignment simply changes what the variable refers to!

What other kinds of things can be assigned to variables in *Python?*

- **Lists**
- **Objects of any class**
- **Files**
- **Dictionaries**
  - Coming up next assignment
- **Functions (!)**
  - Not often used, but possible!
- **All of these are *Mutable***
  - They can (and do) change
  - Multiple variables can refer to same thing
  - All such variable see changes to object, entity

What else can a variable be?

- **None**
- **Not a value!**
- **Results from**
  - Functions that return no result
  - Other unusual circumstances
- **Cannot do arithmetic, comparisons, etc.**
- **Can test for None**

```
if v is None:
   # do something sensible that
   # does not depend upon value of v
```

Questions?

Tuple

- **Definition:– An immutable sequence of values and/or object references**
  - Looks like a list …
    - But with parentheses instead of square brackets
  - P. 363
- **Example**
  - `(1, 5)`
  - `(10, 'word')`
- **Tuples themselves are immutable!**
  - Cannot change what any element refers to
  - Elements may refer to numbers, strings, etc.
    - Unchangeable!
  - Elements may refer to objects
    - Objects themselves may be mutable
    - Identity of objects cannot change!

A little bit like a string:–
If you want to change a tuple, you have to create a copy!

Tuples (continued)

- Tuples can be created by comma-separated sequence of literals
  - `(1, 5)`
  - `(2, 4, 6, 8)`
- Tuples can be created and decomposed by multiple assignment
  - `t = (r, s)`
  - `r, s = t1`
- Tuples can be indexed like lists
  - `t[0]`
  - `t[1]`
- But elements cannot be assigned to
  - ~~`t[0] = 5`~~

Why tuples?

- We already use them:–

```
(W, count) = playOneGame()
```

- Parentheses are optional if unambiguous

```
W, count = playOneGame()
```

- Need for dictionaries, and possibly future homework

Questions?

Useful tidbits

```
import os
```

- **`os.listdir()`**
  - Lists the current directory
- **`os.listdir(path)`**
  - Lists the directory found at **`path`**
- **`os.getcwd()`**
  - Gets the current working directory
- **`os.chdir(path)`**
  - Changes the current working directory to **`path`**
- **`os.mkdir(path)`**
  - Creates a new directory with name **`path`**
  - Absolute or relative to current working directory
- **Lots of other tidbits**

Useful menu items in *IDLE*

- **Path browser**
  - Shows the various directories that *Python* searches to find modules, etc.
  - Listed in order of search
  - See example
- **Class browser**
  - Shows the classes and functions defined in current module
  - Click to get to definition
- **Open Module …**
  - Tries to find and open the module by searching the *path*
  - Opens *Python* modules but not built-in internal modules

Questions?