A quick review of basic Python commands and data structures for INFO-I485

Compiled by Santosh Manicka

This is a quick review of some basic python features that you will need to do your assignments. Of course, this is far from being exhaustive. I have put together some code snippets below. You should be able to run any piece on its own in python version 2.x (*not* 3.x). You can run the code **line-by-line** from the command prompt itself (because Python is an interpreter) or paste whole blocks of code in a file, name it with extension '.py' and then run it from "IDLE" – the popular GUI for python that comes with any python installation. To get you bootstrapped, I have tried to keep this document brief. Anything marked in orange is executable. Also importantly, Python is *indentation-sensitive* by birth – so, please follow the left-indentation that I have used. It doesn't have to be the exact same amount of indentation, however. For example, if you see a tab, it means that there needs to be some left-indentation there; it could be just a single space (I just prefer tabs only for clarity). But if you don't see any indentation, then there shouldn't be any. Again, if you find that directly copy-pasting blocks of code from here to the IDLE command prompt does not work, simply run them from a '.py' file.

```
#Any line that starts with a '#' is a comment, that is Python does not execute it.
```

1. At the beginning of a program you might want to specify that you want to use certain "packages" in order to use their special features that python doesn't automatically provide. Some packages like 'math' and 'array' come with python. Others like 'pygame' and 'numpy' need to be installed. To tell the program that you want to use a package, say, for example:

```
import math as m
#math is the name of the package, m is its alias that is easier to use
```

2. Each package has its features implemented in the form of "modules", which are simply functions to which you can pass parameters and get results in return. You can invoke the modules with a '.' that follows the name of the package (or its alias) followed by the name of the module (modules don't have aliases). For example:

```
m.sqrt(16)   #'sqrt' is the square root module
m.cos(m.pi)  #packages also have constants defined, besides function
```

3. Using conditional logic statements: Note how the else-if is implemented in python – it is 'elif'. Also, notice the colon at the end of the conditions: they are necessary.

```
x=0
```

```
if x==1:
	print "x is 1"
elif x>1:
	print "x is > 1"
else:
	print "x is < 1"
```

4. Loops: Again note the colon at the end of the loop statements.

```
x=0
while x<10:
	x += 1  #the '+= ' which is typical in C also works in python
print x
for i in range(0,10):  #range(0,10) returns a list of values from 0 to 9
	x -= i
print x
```

5. Lists: A list is an indexed container where you can store a variety of elements, that is, you can store integers, decimals, strings etc in the same list.

```
L = [10, 'mad',3.2]  #list containing 3 different types of elements. Note the brackets.
L.append('more')  #append a string to the end of the list
print L[0]  #access an element of the list; indices start with 0, and not 1
print L.index('mad')  #retrieves the 'index' of 'mad' in L
del L[1]  #delete the element stored at index 1
```

6. Tuples: A tuple is very similar to list, in that it is also an indexed container used to store a variety of elements, but it is treated as an *immutable* data-type. You can have a list of tuples, for example, but you can't append to a tuple after it is defined.

```
L = [] #To declare an empty list, you can also say: L = list()
T1 = (2.5, 3.4)  #Note the brackets; they are different from a 'list' definition
L.append(T1)  #appends a tuple to a list
T2 = ((12,14),1.5)  #tuple within a tuple; 2-D coordinates and velocity, for example
print T2[0]  #accessing the contents of a tuple is the same as a list
```

7. Arrays: Python has its own in-built package called "array". Refer to this for documentation - http://docs.python.org/library/array.html. We will not use this package much.

```
import array as arr
x = arr.array('i',[1,4,10])
#'i' specifies that the array will contain integer values. [1,4,10] is the list you want
#the array 'x' to contain
print x
print x[0]  #0 is the first index of the array
print x[2]  #2 is the maximum possible index for this array as its length is 3
```

```
print x*3   #this is NOT a multiplication, rather it replicates 'x' 3 times
```

The much more popular package for using arrays is “numpy”. A nice tutorial is here - http://www.scipy.org/Tentative_NumPy_Tutorial. This is the package we will try and stick to, as it has more and nicer features than the built-in packages.

```
import numpy as np
x = np.array([30,23,100])  #you don't have to mention the datatype (e.g., integer)
print x[1]  #the second element of x: this is how you index an array
y = x*3  #now this is a multiplication
print y  #all elements
print np.where(x == 100)  #retrieve the index of
z = np.zeros(100)  #create an array of length 100 and store 0's in it
l1 = np.log2(2)  #base-2 logarithm of a single number
l2 = np.log2([2,34,456])  #compute the log of an array of numbers
s = np.sum([1,59,98])  #arithmetic sum of an array
print[l1, l2, s]  #print a list of variables
```

There are significant differences between how arrays are treated in the packages ‘array’ and ‘numpy’. For the sake of uniformity, therefore, we will stick to the *numpy* package.

8. Matrices: Generally, matrices are two-dimensional arrays. You can also think of a matrix as an array of arrays. Use numpy to define them:

```
import numpy as np
m = np.matrix([[1,2], [3,4]])  #pass a list of equally-sized lists to define a matrix
print m[0,1]  #retrieves the element in the 0th row and 1st column of the matrix
```

9. Dictionary: It is defined as a set of *keys* and their corresponding *values*. Given a key, you can retrieve its value from a dictionary. Dictionaries are convenient in defining relationships between seemingly unrelated things, alphabetic letters and numbers, for example:

```
Char_to_Num = { 'A':0, 'B':1, 'C':2}  #note the use of curly braces
#the 'key' is defined on the left of the colon, and its 'value' on the right
print Char_to_Num['A']  #retrieves the value of the key 'A'
print Char_to_Num.keys()  #retrieves keys only, and returns them in a list
print Char_to_Num.values()  #retrieves values only, and returns them in a list
print Char_to_Num.keys()[Char_to_Num.values().index(1)]
#retrieve the key for a value
```

10. Strings: They can be considered as arrays and indexed as you index an array

```
s = "python is"
print s[1]  #the second character of the string
s = s + "not a snake"  #the '+' concatenates the two strings
print s
```

11. ASCII values: Each keyboard character has a numerical ASCII value associated with it. Sometimes it is useful to have access to it, as it may be easier to deal with those numbers, rather than the original characters, in your program.

```
print ord('F')  #the ASCII value of upper case F
print chr(76)  #the character whose ASCII value is 76
print chr(ord('s'))  #returns the lower case 's '
```

12. Files: Reading and writing.

```
# First set the working directory, where you want to read and write files
import os
os.getcwd()  #retrieves current working directory
# Change current working directory
os.chdir('/Users/smanicka/Documents/My IU/Fall 2013/Teaching I485/Code')
# Now move on to file operations
f = open("text.txt","r")  #open file for reading; make sure this file exists
lines = f.readlines()  #stores each line of the text in 'lines'
print len(lines)  #number of lines in this file
print lines[3]  #retrieves the 10th line, assuming there are that many lines
print lines[2][5]  #retrieves the 5th character of the 2nd line
f = open("text.txt","r")  #open the file afresh, if you want to read it again
chars = f.read()  #stores all characters of the text, not single lines, in 'chars'
chars[123]  #retrieves the 123rd character of the text.
f = open("text.txt","a")  #open file for writing; append to existing contents
s = 'a string'
f.write(s)
f.close()  #this is required after a write, else you won't see the new contents
```

13. Defining a function: The following should be self-explanatory. Again, please pay attention to the indentation for the statement that follow after the first 'def' statement. To run the following code, create a file called "words.txt" in the working directory, and enter some words in it.

```
def getWords():
	f = open("words.txt","r")
	words = f.readline().split()
	#NOTE: readline() returns a string, while readlines() returns a list.
	#Also, split() works only on a string, NOT on a list, and returns a list.
	return words

# To call a function, you can do the following, for example:
words = getWords()
print words
```

14. 'Random' package: Use this package to pull random numbers from a given range.

```
import random as rnd
```

```
print rnd.randint(0,100)  #randomly pull a number from the range 0 through 100
```

15. Python documentation
    a. Official - http://python.org/doc/
    b. Popular book, "A byte of python - http://www.swaroopch.com/notes/Python
    c. A comprehensive pygame tutorial with exercises - http://tinyurl.com/8wb56z

Exercise

Graph traversal. Take a piece of paper, and make up a small graph (network) with 5 vertices (nodes) in it. Randomly assign a few links or edges between them. Your graph may look something like this. Label the nodes with numbers or strings. Your first task is to represent this graph in Python. You may use any data-structure of your choice: it may be a dictionary with keys standing for node labels and values standing for the neighboring nodes' labels; or two separate lists, one containing all the "from" nodes, and the other containing lists of all the "to" nodes that each "from" node is connected to; or define a *connectivity matrix* (figure out how); or you can get creative. Your second task is to pick up all possible paths, between two chosen nodes, that won't be let to visit any node more than once. Note that since there are only 5 nodes, the maximum number of edges in such paths between any two nodes is only 4 – you can use this as a stopping criterion for any potential path that your program chooses. If you want to be challenged even more, you can randomly assign a direction to each edge in your graph, so your program will be constrained to traverse edges only in the directions specified.