Introduction to Python

BMI 214

1. Installation and use

Getting Python:

Mac OSX:

Python should already be installed but you might want to upgrade (http://www.python.org/download/mac/). If you are going to spend a lot of time in the interpreter environment you should consider downloading ipython. This is a better interpreter shell for python, with very nice things like tab-completion of variable and method names.

PC:

Download at http://www.python.org/download/

Unix:

Python is installed on the elaine, tree, and junior machines. Note: If emacs is your text editor of choice, then download python-mode. It is VERY useful when writing scripts.

Using the python interpreter:

Python can be used in two ways:

1. In the interpreter environment, which reads and executes commands interactively. usage: python
2. Execution of a script. usage: python programFileName.py

2. Python Tutorial

Tips before you start

- Critical to note: Python identifies blocks of code via indentation. Thus, do NOT use spaces, always use tabs. It is extremely sensitive to this, and can cause frustration in finding bugs, if you are not careful.
  For example, there are no curly braces to distinguish blocks of code under if-else statements, simply indentation.
- Also note the colon at the end of 'for' statements, 'else' statements, etc.
- Variables are not declared as they are in high level languages such as C++

Python as a calculator

```
>>> 1 + 2
3
>>> 10 * 2
20
>>> ( 5 * 4 ) + 1
21
>>> 8 / 3                \# Integer division returns the floor:
2
>>> 8 / 3.0         \# float devision returns float:
2.66666666
```

Printing in Python

Note that there is a difference between typing the variable name on the command line, and typing print variable

```
>>> a='Tab	Here'
>>> a
'Tab\tHere'
>>> print a
Tab	Here
```

If you don’t want an endline after the print command, use a comma at the end of the line:

```
>>> for i in [1,2,3]:
...		print i
1
2
3

>>> for i in [1,2,3]:
...	print i,			\# note the comma at the end of the line
1 2 3
```

NOTE:
The lines of code in this document have been copied from an interpreter environment and thus appear with the ‘>>>’ and ‘…’ characters.
To try this, you will type the following lines, pressing ‘Enter’ at the end of each:

```
for i in [1,2,3]:
     print i
```

And the output will appear.
Note that the second line begins with a TAB before the word ‘print’

Lists and Matrices

Lists

```
>>> vec = [ 1, 2 , 3, 4 ]
>>> vec
[1,2,3,4]
>>> vec = range(1, 10)
>>> vec
[1,2,3,4,5,6,7,8,9]
```

You can index lists:

```
>>> vec = range(1, 10)
>>> vec[4]
5
>>> vec[5 :]		\# the colon means continue until end of list
[6,7,8,9]
>>> vec[-1]		\# -1 indexes the last element in list
9
>>> vec[-2]		\# -2 indexes the second to last element in list
8
```

You can append to lists:

```
>>> vec1 = [ ]
>>> vec1.append( 6 )
>>> vec1.append( 7 )
>>> vec1
[6,7]

>>> vec2 = [10, 11, 12]
>>> vec1+vec2
[6,7,10,11,12]
```

You can extend lists:

```
>>> vec1.extend([23])
>>> vec1
[6,7,10,11,12, 23]
```

You can append a list to a list:

```
>>> vec1.extend([[24]])
>>> vec1
[6,7,10,11,12, 23, [24]]
```

You can make lists of repeats

```
>>> vec = [ 0 ]*3
>>> vec
[0,0,0]
```

You can make lists of int, floats, or strings:

```
>>> stringVec = [ "hello", "world" ]
```

The len function tells you the length of the list:

```
>>> len(stringVec)
2
```

The sort function will sort a list:

```
>>> vec = [ 5, 4, 1, 2 ]
>>> vec.sort()
>>> vec
[1,2,4,5]
```

Matrices

You can use nested lists to make matrices, 3D arrays, etc...

```
>>> matrix = [[ 1, 2, 3, 4] , [ 5, 6, 7, 8]]
```

You can index the matrix:

```
>>> matrix[0]
[1,2,3,4]
>>> matrix[0][2]
3
```

Control Structures

If

Consider the script testPositive.py:

```
#import the sys module
```

```
import sys
#sys.argv is a list of space delimited command line inputs
#sys.argv = [ “testPositive.py", "5"]
#since it is a list of strings we convert it to an int:
x = int(sys.argv[1] )
print ""
if x < 0:
        print "negative"
elif x == 0:
        print "zero"
else:
        print "positive"
print ""
print "sys.argv[0] is:", sys.argv[0]
print "sys.argv[1] is:", sys.argv[1]
```

We run the script by typing ‘python testPostitive.py 5’

For

You can use for to iterate through lists:

```
>>> for i in range(10):
...         print i
```

You can either iterate through a list directly:

```
>>> vec = ['a', 'b', 'c', 'd','e']
>>> for letter in vec:
...             print letter
```

Or iterate through list without defining it as a variable:

```
>>> for i in ['a', 'b', 'c', 'd','e']:
...             print i
```

Or iterate through the list by indexing:

```
>>> for i in range(len(vec)):
...             print vec[i]
```

Use break to exit loops:

```
>>> for i in range(10):
...             print i
...             if i > 5 :
...                     break
```

Modules

As seen above we had to import a module called sys to get command line arguments. Modules are scripts that contain python definitions. Lets look at the script testPositive.py that we looked at before. We can make it a module by making a definition:

```
def CheckSign(x):
        if x < 0:
                result= "negative"
        elif x == 0:
                result = "zero"
        else:
                result = "positive"
        return(result)
```

We can then import the module in the command line or within another script.

```
>>> import testPositive
>> > answer = testPositive.CheckSign( -5)
```

If we are using many definitions in a module then we might not want to bother typing the module name before calling each definition. In this case import all the definitions from the module at the beginning of the script.

```
>>> from testPositive import *
>>> answer = CheckSign( -5 )
```

Modules must be imported at the beginning of each script. Python comes with many modules that you will want to use often. Some common ones are:

string Although python on its own allows for some string manipulation, the string module allows for more advanced functionality such as find() , replace() , and join() . (see below)

math This module provides access to many mathematical functions (ex. ceil, floor, abs, sqrt, log)

sys This module provides access to variables used or maintained by the interpreter.

os This module allows you to interact with the operating system within a script. (ex. os.system("rm fileName") to remove a file in the current directory)

Strings

You can index strings:

```
        >>> classString = "biomedical informatics"
        >>> len(classString)
        22
        >>> classString[ 3 : ]
        "medical informatics"
```

you can concatenate strings using +

```
        >>>"abc"+"def"
        "abcdef"
```

The string module

There are many functions that help with string

```
>>> classString = "biomedical informatics"
>>> classString.find( "info" )
11

>>> classString.replace( "informatics","computation" )
"biomedical computation"

>>> classVec = classString.split() #can aslo split on \t, \n etc..
>>> classVec
["biomedical","informatics"]

>>> print " ".join(classVec)
"biomedical  informatics"

>>> a = 'note spaces at end    '
>>> a
'note spaces at end    '
>>> b=a.strip()
>>> b
'note spaces at end'
```

Conversion

As seen previously you can convert a string to a float or int:

```
>>> numString = "3.0"
>>> float( numString )
3.0
```

conversely you can also convert a float or int to a string:

```
>>> numString = str( 3.0 )
>>> numString
"3.0"
```

File IO

To open a file (read = ”r”, write = ”w”, append = ”a”):

```
    >>> inFile = open("foo.txt","r")
```

The command read() will read the entire file to a string

```
    >>> fileString = inFile.read()
```

The command seek() will reset the position in the file to the beginning

```
    >>> inFile.seek(0)
```

You can read one line of the file at a time:

```
    >>> line1 = inFile.readline()
    >>> line2 = inFile.readline()
```

or you can read every line into a list:

```
    >>> inFile.seek(0)
    >>> lineVector = inFile.readlines()
```

You can iterate through the file this way:

```
    >>> inFile.seek(0)
    >>> for line in inFile.readlines():
    ...         print line
```

Alternatively:

```
    >>> for line in inFile:
    ...        print line
```

You will notice in the last two examples it will print a double space between the lines. That is because line has a endline character “n” and print also adds a line to the end.

Dictionaries

Dictionaries are sets of key:value pairs (similar to hash table in other languages)

```
>>> classDict = {"BMI" : "biomedical informatics", "CS" : "computer science"}
>>> classDict[ "BMI" ]
"biomedical informatics"
```

You can get a list of keys at:

```
    >>> classDict.keys()

    >>> for key in classDict.keys():
               print key, classDict[key]

                           #the comma separates items to be printed by a space.
```

Functions

Within the same script

To define a function within a python script, we use def:

```
myScript.py

def printStuff(firstName, lastName):
    print "First name is: ", firstName
    print "Last name is: ", lastName
    return firstName+lastName
#----------------------------------------
#        MAIN SECTION

import sys
print "This is the main section"
firstName=sys.argv[1]
lastName=sys.argv[2]
a=printStuff(firstName, lastName)            # function printStuff returns a string
print a
```

From another script

To call a function defined in another file, use import:

myScript.py

```
def printStuff(firstName, lastName):
        print "First name is: ", firstName
        print "Last name is: ", lastName
        return firstName+lastName
```

myScript2.py

```
import sys
import myScript

print "This is the main section"
firstName=sys.argv[1]
lastName=sys.argv[2]
a= myScript.printStuff(firstName, lastName)
```

And then call 'python myScript2.py John Doe'

OR

myScript3.py

```
import sys
from myScript import *

print "This is the main section"
firstName=sys.argv[1]
lastName=sys.argv[2]
a= printStuff(firstName, lastName)
            #no need to preface function with myScript.
```

And then call 'python myScript3.py John Doe'

In project1, you may have a variable x=0.03000000002 that you want to print out as 0.03. You'll find that printing to the screen and writing to files have slightly different behavior in Python. To write to file:

```
file = open("temp.txt", "w")
x=0.0300000002
file.write("%.2f" % x)
```

Additional useful Python stuff:

- **Check out list comprehension. Very cool.** A good way to learn about this can be found at: http://www.diveintopython.org/native_data_types/mapping_lists.html
- Filtering lists: http://www.diveintopython.org/power_of_introspection/filtering_lists.html