Python Cheat Sheet

Miscellaneous useful notes

1. Text after a pound sign (#), as well as text between sets of three single quotes, `'''`, are **comments**, and are ignored by the code.

2. Text in parentheses is evaluated by Python prior to other text, in a manner identical to how parentheses are treated in math.

3. Most Python IDEs (i.e., most programs used when writing Python), including Spyder and PyScripter, have to environments to work in - the **editor** and the **command window**. The editor is for writing large pieces of code that are eventually saved as Python files, i.e., text in the editor represents what will become your program. The command window is for writing quick pieces of code to see them executed, usually one line at a time.

4. The most useful extensions to Python are NumPy, SciPy, and matplotlib. All of the functionality of these can be accessed in Python by using the **import** command, as in the following code.

```
import numpy as np
import scipy as sp
import matplotlib as mpl
#The 'as' keyword just let's you call these packages by a shorter name when
#using them, as in the line of code below.
print np.abs(-5)#This would print 5
```

Resources

The following are Python distributions - after you download and install them, Python and most of its numerous useful extensions will work on your computer.

WinPython http://winpython.sourceforge.net/. Windows XP/7/8.

Anaconda https://store.continuum.io/cshop/anaconda/. Linux: 64-bit and 32-bit. Windows: 64-bit and 32-bit. Mac OS X: Intel 64-bit.

Additionally, **Codeacademy** is a great place to get started with Python, http://www.codecademy.com/. It has simple step-by-step lessons that will familiarize you with the syntax and "feel" of the language (and of programming in general).

Data types

The most important data types in python are **int**, **float**, **str**, **bool**, **list**, and **numpy.array**. Lists are containers of any miscellaneous combination of the other data types. Arrays are containers of either integers or doubles. Lists are easier to work with than arrays, but code that uses them is much slower. An example of each data type is below.

```
import numpy as np
int1=5; int2=-5; #int1 and int2 are ints, i.e., integers.
float1=5.0; float2=23.2; #float1 and float2 are floats, i.e., decimal numbers.
str1='Hello'; str2='1' #str1 and str2 are strings, i.e., groups of characters.
bool1=False; bool2=5>2 #bool1 and bool2 are bools, i.e., things that are True or False.
list1=[1,2.56,"Hello",True] #list1 is a list. Note multiple data types.
array1=np.array([2.0,-0.5,2.56,2.2]) #array1 is an array. Note all floats.
'''
To access different elements in a list or array, use [element number].
The first element has element number=0.
For example, the line below is a boolean with value True.
'''
list1[1]==array1[2]
```

Note the two different types of types of "equals" that are used in the above code. A single-equals, =, is used for assignment of a value. A double-equals, ==, is used for comparison, just as <= or >= would be.

Functions

A **function** is anything that takes 0 or more data types as an input, and performs a task (based on those inputs). Several functions are already built into Python and its extensions. For instance, `numpy.sin(x)` is a function built into numpy that finds the sine of the input `x`.

You can define functions in Python using the keyword `def`. The format of a function is as follows:

```
def functionName(inputVariable1,inputVariable2,...):
    body#Code manipulating input variables
    return outputVariableName#This line is not required for all functions
```

Some example functions are defined below. Note the use the colon and of tabs, required as an integral part of Python syntax. Also note the use of the keyword `return`- whatever comes after this word is the output of the function.

```
#Prints the word hello
def sayHello():
    print "Hello."

#Prints x twice
def printTwice(x):
    print x
    print x

#Outputs the sum of a and b
def plus(a,b):
    return a+b

#Takes two strings, and combines them into a single string
def concatenate(a,b):
    return a+b

'''
Note that the scripts for plus() and concatenate() are the same!
This is because Python is smart enough to often let you not worry about which
data type you are being used. E.g.,
a=2;b=3;a+b                 --->    Python gives 5
a='Hello ';b=' world!';a+b  --->    Python gives "Hello world!"
'''
```

Conditionals and logical operators

You will frequently want your functions to behave differently depending on what conditions your inputs satisfy. **Conditionals** are statements that tell the functions to do just this. The keywords that signal them are `if`, `elif`, and `else`. As with functions, a colon and tabs are required as part of the syntax of a conditional. The format of a conditional is as follows:

```
if condition1:#condition1 is something that can evalute to True or False:
    body1#Code to do if condition1==True
elif condition2:#condition2 is something that can evalute to True or False:
    body2#Code to do if condition2==True
else:
    body3#Code to do if none of the conditions in the if and elif's were True

#There can be any number of elif statements, including 0. There can be 0 or 1 else statements.
```

Some trivial functions that apply this concept are given below. Note the use of the **logical operators** `and`, `or`, and `not`. These behave exactly as you would expect.

```
#Prints "a is 1!" if a==1
def printIfOne(a):
    if a==1:
        print "a is 1!"

#True if a==1, False otherwise
def decideIfOne(a):
    if a==1:
        return True
    else:
        return False

#True if a==1 or a==2, False otherwise. Uses elif.
def decideIfOneOrTwoVersion1(a):
    if a==1:
        return True
    elif a==2:
        return True
    else:
        return False

#True if a==1 or a==2, False otherwise. Uses or.
def decideIfOneOrTwoVersion2(a):
    if a==1 or a==2:
        return True
    else:
        return False

#Returns True if at least one of b or c is bigger than a, False otherwise.
#Uses not, and.
def aNotTheBiggest(a,b,c):
    if not (a>=b and a>=c):
        return True
    else:
        return False
```

(As a sidenote, a shorthand for `not (a==b)` is `a!=b`).

Loops

You will frequently want your functions to perform a given task multiple times. Loops cause a chosen portion of your program to do this. There are two types of loops - **for loops** and **while loops**. For loops cause a portion of your program to repeat for a set number of iterations, and while loops cause a portion of your program to repeat while a given condition is met. The format for each of these loops is as follows:

```
#FOR LOOP FORMAT
for var1 in list1:
    body
#e.g., if list1=[1,3,5], then body executes -
#First with var1=1, then with var1=3, and finally with var1=5.

#WHILE LOOP FORMAT
while condition1:
    body
#The while statement acts just like an if statement, except that when the body
#is finished, it repeats the if statement, until condition1 is False.
```

Some trivial blocks of code that apply these concepts are given below. Note the use of the `range`function. This is a very useful built-in Python function, which returns a list as follows: `range(num)=[0,1,2,...,num-1]`.

```
#FOR LOOP FORMAT
for var1 in list1:
    body
#e.g., if list1=[1,3,5], then body executes -
#First with var1=1, then with var1=3, and finally with var1=5.

#WHILE LOOP FORMAT
while condition1:
    body
#The while statement acts just like an if statement, except that when the body
#is finished, it repeats the if statement, until condition1 is False.
```

Python also has a very handy shorthand that can be used in place of many for loops when workin with lists. It is demonstrated in the code below. Sidenote: note the use of the `%` operator in the code.

```
#Note: The % operation is called 'modulo'
#a%b gives the remainder of a/b.
#E.g., 11/3 = 3 remainder 2, so 11%3 = 2.

list1=[1,2,3,4,5,6,7,9,10,11,12]

listLists1=[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]

list1Odd=[item for item in list1 if item%2==1]#Get all odd numbers in list1

listLists1First=[item[0] for item in listLists1]#Get 1st number of each sublist in listLists1

#The above code gives:
#list1Odd=[1,3,5,7,9,11]
#listLists1First=[1,4,7,10]
```

Extra - List manipulation and array manipulation

As mentioned earlier, the two most important "big" data types in Python are **lists** and **numpy arrays**. (These are big in the sense that they contain other data types). The default operations (+, ∗, etc.) act differently with each of these data types. With lists, they specify concatenation (patching together), whereas with arrays, they specify element-wise arithmetic. However, both either operation can be used with both data types. The code below demonstrates how to do various operations with each.

```python
import numpy as np

array1 = np.array([1,2,3])
array2 = np.array([4,5,6])

list1 = [1,2,3]
list2 = [4,5,6]

'''Element-wise operations'''
#Added: Gives [5 7 9]
#Multiplied: Gives [4 10 18]
#MultipliedBy4: Gives [4 8 12]
#Subtracted: Gives [-3 -3 -3]

arraysAdded = array1 + array2
arraysMultiplied = array1 * array2
array1MultipliedBy4 = array1 * 4
arraysSubtracted = array1 - array2

listsAdded = [list1[item] + list2[item] for item in list1]
listsMultiplied = [list1[item] * list2[item] for item in list1]
list1MultipliedBy4 = [4 * item for item in list1]
listsSubtracted = [list1[item] - list2[item] for item in list1]

'''Concatenation'''
#SelfConcatenated: Gives [1 2 3 1 2 3]
#Concatenated: Gives [1 2 3 4 5 6]

array1SelfConcatenated = np.append(array1,array1)
arraysConcatenated = np.append(array1,array2)

list1SelfConcatenated = list1 * 2
listsConcatenated = list1 + list2
```

There are also some shorthands for initializing lists and arrays to be between two numbers, with a given stepsize or number of steps between each number. Most of these are variations on the function `range`. A demonstration is below.

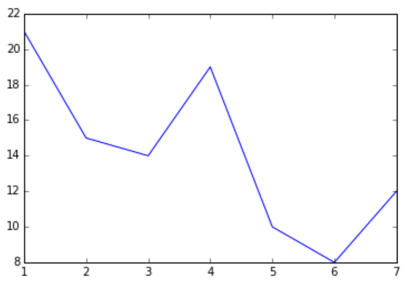
```python
import numpy as np

'''range() and np.arange()'''
#1 argument: Start = 0, End = (argument-1).
#2 arguments: Start = 1st argument, End = (2nd argument-1).
#3 arguments: Start = 1st argument, End = (2nd argument-1), Step = 3rd argument.
'''np.linspace()'''
#Start = 1st argument, End = 2nd argument, Number of steps = 3rd argument.

list1 = range(10)
list2 = range(4,10)
list3 = range(4,10,3)

#list1 = [0,1,2,3,4,5,6,7,8,9]
#list1 = [4,5,6,7,8,9]
#list3 = [4,7]

array1 = np.arange(10)
array2 = np.arange(4,10)
array3 = np.arrange(4,10,3)
array4 = np.linspace(4,10,5)

#array1 = np.array([0,1,2,3,4,5,6,7,8,9])
#array2 = np.array([4,5,6,7,8,9])
#array3 = np.array([4,7])
#array4 = np.array([4.0,5.5,7.0,8.5,10.0])
```

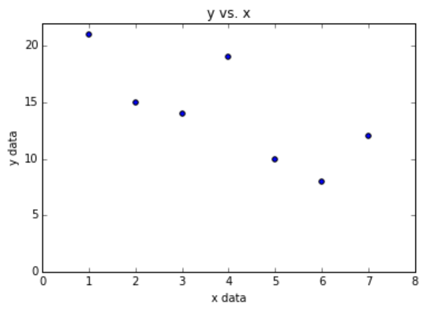
Extra - Plotting

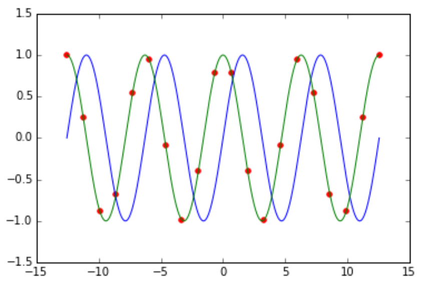
Plots are usually made using matplotlib. The code below demonstrates how to make line plots and scatter plots of two lists or arrays of data against one another, as well as how to plot functions. How to display and save the plots is also demonstrated.

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

x = [1,2,3,4,5,6,7]
y = [21,15,14,19,10,8,12]

#Line plot: y against x.
#Show the plot.
plt.plot(x,y)
plt.show()

#Clear the plot
plt.clf()

#Scatter plot: y against x.
#Label plot.
#Change minimum/maximum x/y that is shown on plot.
#Save the plot.
plt.scatter(x,y)
plt.title('y vs. x')
plt.xlabel('x data')
plt.ylabel('y data')
plt.xlim(xmin=0,xmax=8)
plt.ylim(ymin=0,ymax=22)
plt.savefig('myPlot.png')

#Clear the plot
plt.clf()

#Plot y = sin(t) and y = cos(t). Scatter y = cos(t).
#Show the plot.

#Make tPlot dense (1000 points), so plots will look continuous.
tPlot = np.linspace(-4*np.pi,4*np.pi,1000)
#Make tScatter not that dense, so will be able to distinguish scatter points.
tScatter = np.linspace(-4*np.pi,4*np.pi,20)
y1 = np.sin(tPlot)
y2 = np.cos(tPlot)
y3 = np.cos(tScatter)
plt.plot(tPlot,y1)
plt.plot(tPlot,y2)
plt.scatter(tScatter,y3,color='red')
plt.show()
```

The plots generated by the above code are below. Only the middle one was saved by the code as a picture. The other two were shown in the IDE being used.


22
20
18
16
14
12
10
8
1
2
3
4
5
6
7


y vs. x
y data
x data


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−15
−10
−5
0
5
10
15