Introduction to Python

This IPython notebook provides an introduction into some of the basics of python and the coding idioms of Python. We will start by reviewing the different types of data that can be stored in Python and proceed to other topics from there. While you work through this notebook it would be wise to either be playing with the code in the notebook itself or in a ipython console that you open on your own. I will provide some comments intended for the reader that are denoted by a # in Python. Remember that while you are learning Python and even once you master Python that Google will be your best friend. If you ever have a question about how to do something then you will almost certainly be able to find an answer online. Following the information in this notebook, there will be several problems that you will be asked to turn in.

Types of Data in Python

This section will introduce you to several different types of data that you will frequently use in Python. We will touch on strings, integers, floats, and lists (Note: There are complex numbers as well, but they are very similar to floats and I also suggest learning a little about sets and dictionaries). Each of these will have different uses and you will need them to complete future labs. As a note prior to beginning. In python, x = _ , denotes x as whatever fills in the blank space after the equal sign. This is known as being an identifier. You can use whatever variable/word you want with the exeception of beginning your identifier with a number and for several words that python has claimed for itself (and, del, from, not, while, as, elif, global, or, with, assert, else, if, pass, break, except, import, print, class, exec, in, raise, continue, finally, is, return, def, for, lambda, try). Below we define num as an identifier for the object 1. Notice when we tell python to print the identifier num that it returns 1. Another important to note is that python begins indexing at element 0. If you want the first element from a list or string then it will be denoted as the 0th element.

In [23]:
```python
num = 1
print(num)  # This will return what num is tied to (1)
```
```
1
```

In [24]:
```python
testlist = [0, 1, 2, 3, 4]
print(testlist[0])  # Notice this gives us the first element of testlist.
```
```
0
```

Strings

Strings are a list of characters(includes numbers and symbols) in a certain order. These are declared in python by using quotation marks (', ", or """). Notice below that when we ask python to print what type 'hello' is that it returns type str. This is letting us know that 'hello' is a string.

```
In [3]: print(type('hello'))
        'hello'

        <type 'str'>

Out[3]: 'hello'
```

There are several operations that can be done to strings. Typically strings are useful for printing updates or information from the code, but they can be used for other things as well (such as creating urls to direct python to). For example, when printing, you can print multiple lines within one print statement by using \n. You can also concatenate(add). Look at the difference between the examples below.

```
In [27]: str1 = 'welcome'
         str2 = 'to'
         str3 = 'bootcamp'
         print(str1 + str2 + str3)  # Here we just print the strings concatenated togeth

         welcometobootcamp

In [29]: print(str1 + ' \n'+str2 + ' \n'+str3)# Here we concatenate&printthe strings and

         welcome
         to
         bootcamp
```

We can also slice(pull specific elements from) strings. Look at the examples below (The %s is a place holder for what comes after %) :

```
In [36]: practicestr = 'abcdefghi'
         print('first element of practicestr is "%s"' % practicestr[0])

         first element of practicestr is "a"
```

We can also access a range of elements from strings in the following way.

```
In [39]: pracstr = 'abcdefghi'
         print(pracstr[0:])  # The : means everything afterwards.  This will print whole
         print(pracstr[5:])  # This will print from element 5 on

         abcdefghi
         fghi
```

Now see what the command [::-1] does after the string.

```
In [41]: pracstr = 'abcdefghi'
         # Write your practice command here
```

There are other things that you can do with strings, but this should suffice for now. A couple of the other cool things are included below (The actual command is after the %). If you want more instruction on any of these, ask questions.

```
In [73]: string = 'thisisasentencestring'
         capitallets = 'ABCDEFGH'
```

```
print('number of a\'s in string = %s') %string.count('a')
print('string in all upper case is %s') %string.upper()
print('capitallets in all lower case is %s') %capitallets.lower()
print('replace D in capitallets with another A is %s') %capitallets.replace('D'
print('a is in the %s element of string') %string.find
```

```
number of a's in string = 1
string in all upper case is THISISASENTENCESTRING
capitallets in all lower case is abcdefgh
replace D in capitallets with another A is ABCAEFGH
```

Integers and Floats

Integers are exactly what they sound like. Integers are all positive and negative integers; for you math folk it is $\mathbb{Z}$. Floats are all of the real numbers, $\mathbb{R}$. It is important to note the difference in programming between integers and floats because their operations will do different things. To declare a number as a float, you either need to tell the computer it is a float via command or add a decimal place. See below:

```
In [46]: type(1)  # Just a normal integer will be recorded as an integer

Out[46]: int
```

```
In [45]: type(1.0)  # Just adding a decimal place is the easiest way to declare a float

Out[45]: float
```

```
In [47]: type(float(1))  # but you can tell python that something is a float.

Out[47]: float
```

```
In [63]: print(int(5.5))

         5
```

```
In [64]: print(float(5))

         5.0
```

I told you that they were going to be a little different so let me show you the difference. We obviously know that 2/4 should be equal to .5, but lets see what happens when we try it with integers.

```
In [48]: 2/4

Out[48]: 0
```

Integer division gave us 0 for 2/4. This is the wrong answer. The explanation is actually pretty simple. The division algorithm says:
Let a, b be integers with $b > 0$ then there exist unique integers q and r such that

$a = bq + r$ and $0 \leq r < b$

Note that r is actually the remainder. Integer division simply returns the q from this equation.

Now if we do this division in floats, we will get the answer that we are looking for.

```
In [49]: 2.0/4.0

Out[49]: 0.5
```

Integer operations certainly have their place, but typically you will want to be using float division. If at least one of the numbers is a float then the output will be returned as a float. There is also a small cheat you can put at the beginning of your file that will declare all of your division as float division for your whole script. That cheat is to write from __future__ import division at the beginning of your script. That will make all division float division.

```
In [54]: from __future__ import division
         4/3

Out[54]: 1.3333333333333333
```

The rest of the operations should work pretty similarly between the two, but it is always better to be safe and work with floats if you want to be in the real numbers. Another operation that tends to be very useful is modulus. You do modulus with %. The modulus will return the r from the division algorithm. This will be useful in many instances. For example, if you need to know whether a number is even or odd then you can check the number modulus 2.

```
In [57]: 4%2  # gives us 0 so 2 must divide evenly into 4

Out[57]: 0

In [58]: 5%2  # gives us 1 so 2 doesn't divide evenly into 5

Out[58]: 1

In [59]: 143621362 3%2  # gives us 1 so 2 doesn't divide evenly into that number

Out[59]: 1
```

In many programming languages the exponent is denoted as ^, but this is not true in python. If you want to take something to a power then you need to use **. See below:

```
In [60]: 3**2

Out[60]: 9

In [61]: 11**4

Out[61]: 14641
```

Lists

Lists are extremely useful for keeping track of information. A list can contain floats, integers, strings or any combination of them. A list is declared by using the square brackets, []. You can append/remove new items to/from a list.

```
In [90]: practicelist = [1, 2, 3, 4]
         practicelist

Out[90]: [1, 2, 3, 4]
```

```
In [91]: practicelist.append(5)  # append 5 to our list
         practicelist

Out[91]: [1, 2, 3, 4, 5]
```

```
In [92]: practicelist.remove(1)  # remove 1 from our list
         practicelist

Out[92]: [2, 3, 4, 5]
```

You can also replace elements within a list.

```
In [95]: anotherlist = ['a', 2, 3, 4]
         print('this is the first element of another list', anotherlist[0])

         ('this is the first element of another list', 'a')
```

```
In [97]: anotherlist[0] = 1
         print('now this is the first element of another list', anotherlist[0])

         ('now this is the first element of another list', 1)
```

Taking slices of lists works the same way as taking slices of strings.

```
In [99]: list1 = [1, 2, 3, 4, 5, 6, 7]
         list1[2:5]  # take elements 2 through 5

Out[99]: [3, 4, 5]
```

We can also concatenate lists just like we did with strings.

```
In [93]: list1 = [1, 2]
         list2 = ['a', 'b']
         list1 + list2

Out[93]: [1, 2, 'a', 'b']
```

In addition to being able to do these things that you were able to do in a string, lists allow you to find the max and min (among other things) within the list.

```
In [108]: list1 = [1, 2, 36, 256, 2562, 56]
          print('The max is %s' %max(list1))
          print('The min is %s' %min(list1))

          The max is 2562
          The min is 1
```

One of the built in commands to build lists is with the range command. Below we will build a list of numbers from 0 to 9 using range in several different ways.

```
In [100]: range(10)  # if you just input an integer it will build up until that number

Out[100]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [101]: range(0,10,1)  # you can tell it to start at 0 and go to 10 taking 1 unit steps

Out[101]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [102]: range(1,10)  # we can go from 1 to 9 as well by specifying to start at 1

Out[102]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Conditionals, Loops, and Booleans

Conditional statements, loops, and booleans permit a programmer increased flexibility. These permit you to give your computer specific commands that save you a lot of coding. For example, if you want to remove all of the even numbers from a list then you can say something like below:

Also some symbols that will be useful:

< less than

> greater than

<= less than or equal

> = greater than or equal

!= not equal

== equal to

```
In [113]: list1 = range(25)
          print(list1)

          [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
         22, 23, 24]
```

```
In [112]: list2 = [list1[n] for n in list1 if n%2!=0]  # This removes all even numbers
          print(list2)

          [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

Booleans are just values tied to certain expressions that evaluate to either true or false. If and while statements are evaluated based on these values. For example, below we said 5 < 3 and the computer says False because that isn't true

```
In [117]: 5 < 3

Out[117]: False
```

The conditional statements that you should be most familiar with are "if/elif/else" and "while". They are used exactly how you would imagine that they are used. You want to tell the computer if something is true then do this, else do this other thing. You could also tell your computer while this condition is true then continue doing what I told you to do. Below are two examples (Note where the indentations (done by inserting 4 spaces) are):

```
In [118]: x = 3
          if x < 3:
              print('x is less than 3')
          elif x < 4:
              print('x is less than 4')
          else:
              print('x is not less than 4')

          x is less than 4
```

```
In [115]: x = 1
          while x < 7:
              print('x is still less than 7')
              x = x + 1  # Increase x by 1

          x is still less than 7
          x is still less than 7
          x is still less than 7
          x is still less than 7
          x is still less than 7
          x is still less than 7
```

For conditional statements, if you are asking whether something is equal you need to use two =. This is because python thinks one = means that you are setting something equal to that, but you are asking whether it is or isn't. See example:

```
In [124]: x = 6
          if x == 6:
              print('told ya so :)')

          told ya so :)
```

Looping is a very similar to a while loop, but you give it a specific number of times it should do something. For example if we had a list and we wanted to square every element, we could use a for loop.

example if we had a list and we wanted to square every element, we could use a for loop.

```
In [121]: list1 = [1, 2, 3, 4, 5, 6]

          for i in range(len(list1)):  # len(list1) tells us how many elements we need tc
              list1[i] = list1[i]**2  # The ith element of list1 = (ith element of list1)

          print('list1 squared is %s' %list1)

          list1 squared is [1, 4, 9, 16, 25, 36]
```

Developing a Function and Exploring Modules

Python Modules

One of the biggest benefits of python is the fact that it is open source. This means that there are almost constanly individuals contributing to python to build modules. Modules are just files that contain python code that you are able to use. Some of the modules that you will use very frequently are: math, numpy, pandas, byumcl, and other modules you might write for yourself. Although you will use these most, if you ever think that something would be nice to be able to do then most likely someone has written a module that does that task. Once again, google will be your best friend in finding these.

You need to import modules into your session to be able to use them. You need to do all of your imports at the very beginning when you are writing a script. These imports are done like this:

```
In [106]: import numpy as np  # the 'as np' part creates a nickname for functions from th
          import math
          import os
          from filename import *  # will import functions that you have written in anothe

          """
          Here you would write the rest of your script
          """

          x = math.pi
```

You have to refer to the module you just imported in order to call an object from that module. For example: We wrote math.pi in the code above. This tells python to look in the math module and call the function pi. When we write things like the "import numpy as np", we are giving numpy a nickname and we can write np.numpy_function instead of writing numpy.numpy_function. It is also helpful to try help(module) to see if you can find documentation. You can also use module.object? to open the documentation for a specific object in a module.

Defining Functions

Although python provides you with many built in functions, sometimes you need to write functions that perform very specific actions. Luckily, python allows you to write your own functions. This is actually very simple. All you need to do is follow is give your function a name, ask for inputs, write the function, and return something. Indentation is important here as well. See example:

In [125]:
```
def square(x): # We give the function the name square and say we pass in x
    '''
    Really pretty documentation that you should write all the time

    Parameters
    ----------
    ....

    Outputs
    -------
    ...
    '''

    output = x * x  # define some variable that we will be the output of our fu
    return output  # return the variable output
```

In [126]:
```
square(45)
```

Out[126]:
```
2025
```

Writing Scripts and the Zen of Python

Writing a Script

Writing a script is pretty simple, so I will try and go over it pretty quick. A script should start with all of your imports and then you write your code. It should be neat and you should leave one clean line at the end of your script or the code monster will come and eat you. Example of a script:

In [128]:
```
import math
import os
from __future__ import division

x = 2.0

y = x * math.pi

finalstuff = square(y)
```

After you have written a script then you need to enter your ipython console. From there, direct your computer to the directory where the script is stored and type "run script.py" That's it for running scripts.

Zen of Python (PEP 20)

The Zen of Python is a set of basic rules that one should follow when writing python code. They are important to follow as they will make your code readable and will bring good code karma to you and your family. Anytime that you forget these rules, type "import this" into your console.

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

```
In [129]: import this
```

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Style Guide for Python (PEP 8)

PEP 8 is too long to put into this document, but I would highly suggest reading through it. The suggestions in PEP 8 are very helpful in making your code readable and in line with the Zen of Python. I will include a few of the tips that I find important.

Indentation: Use 4 spaces per indentation level. Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following considerations should be applied; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

Maximum Line Length: Limit all lines to a maximum of 79 characters.

Blank Lines: Separate top-level function and class definitions with two blank lines. Method definitions inside a class are separated by a single blank line. Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Whitespace in Expressions: Avoid extraneous whitespace in the following situations: Immediately inside parentheses, brackets or braces; Immediately before a comma, semicolon, or colon, Immediately before the open parenthesis that starts the argument list of a function call, Immediately before the open parenthesis that starts an indexing or slicing, More than one space around an assignment (or other) operator to align it with another. If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgement; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator. For example:

Yes

In [130]:
```
x = 0
```

```
x += 1
x = x + 1
x = x**2 + 2
test = x*x + 2*x
y = (x+2) * (x-3)
```

No

In [ ]:
```
x=x+1
x +=1
x = 2 * x + 1
y = (x + 2) * (x - 3)
```

Comments: Comments that contradict the code are worse than no comments. Always update comments! Comments should be complete sentences.

We are hoping to edit your code and give you tips on how to make it stay consistent with PEP 8 and PEP 20. We recommend using the sublime_linter in Sublimetext 2 to stay in line with PEP 8.

Problems:

All scripts submitted should obviously be in line with PEP 8 and PEP 20 for this assignment because that is one of the topics covered. Include comments.

Problem 1: Write a script that generates the following:

i) A list with all of the elements from 0 to 100

ii) A list with all of the even elements from 0 to 100

iii) A list that contains all of the even numbers starting at 100 and going to 0

Problem 2: Write a script that creates a piecewise function f(x) = y that satisfies the following:

i) When x < 0 then y = 0

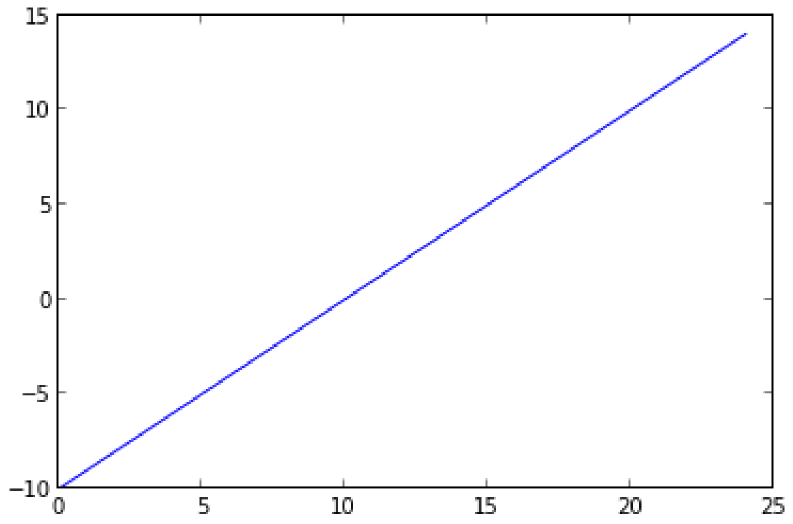
ii) When 0 <= x <=5 then y = x*x

iii) When x > 5 then y = 25

iv) Now copy the following code to create a plot

In [132]:
```
import matplotlib.pyplot as plt
# Define function in here
test = range(-10, 15, 1)
plt.plot(test, [f(i) for i in test])  # Note this isn't your actual function be
plt.show()
```

```
Out[132]: [<matplotlib.lines.Line2D at 0x7070828>]
```



Problem 3: If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Write a script that prints the sum of all the multiples of 3 or 5 below 1000.

```
In [ ]:
```