Get Started with Python

Rajath Kumar M.P.

Department of Electronics and Communication Engineering
RNS Institute of Technology, Bangalore

rajathkumar.exe@gmail.com

Contents

1 Python

1.1 Introduction

Python is a modern, robust, high level programming language. It is very easy to pick up even if you are completely new to programming.

1.2 Installation

Mac OS X and Linux comes pre installed with python. Windows users can download python from https://www.python.org/downloads/ .

To install IPython run,

```
$ pip install ipython[all]
```

This will install all the necessary dependencies for the notebook, qtconsole, tests etc.

1.2.1 Installation from unofficial distributions

Installing all the necessary libraries might prove troublesome. Anaconda and Canopy comes pre packaged with all the necessary python libraries and also IPython.

Anaconda Download Anaconda from https://www.continuum.io/downloads

Anaconda is completely free and includes more than 300 python packages. Both python 2.7 and 3.4 options are available.

Canopy Download Canopy from https://store.enthought.com/downloads/#default

Canopy has a premium version which offers 300+ python packages. But the free version works just fine. Canopy as of now supports only 2.7 but it comes with its own text editor and IPython environment.

1.3 Launching IPython Notebook

From the terminal

```
ipython notebook
```

In Canopy and Anaconda, Open the respective terminals and execute the above.

1.4 How to learn from this resource?

Download all the ipython notebooks from this repository https://github.com/rajathkumarmp/Python-Lectures

Launch ipython notebook from the folder which contains the notebooks. Open each one of them

```
Cell > All Output > Clear
```

This will clear all the outputs and now you can understand each statement and learn interactively.

1.5 License

2 The Zen Of Python

```
In [1]: 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!
```

3 Variables

A name that is used to denote something or a value is called a variable. In python, variables can be declared and values can be assigned to it as follows,

```
In [2]: x = 2
        y = 5
        xy = 'Hey'

In [3]: print x+y, xy

7 Hey
```

Multiple variables can be assigned with the same value.

```
In [4]: x = y = 1

In [5]: print x,y

1 1
```

4 Operators

4.1 Arithmetic Operators

Symbol	Task Performed
+	Addition
-	Subtraction

Symbol	Task Performed
/	division
%	mod
	multiplication
//	floor division
	to the power of

```
In [6]: 1+2

Out[6]: 3

In [7]: 2-1

Out[7]: 1

In [8]: 1*2

Out[8]: 2

In [9]: 1/2

Out[9]: 0
```

0? This is because both the numerator and denominator are integers but the result is a float value hence an integer value is returned. By changing either the numerator or the denominator to float, correct answer can be obtained.

```
In [10]: 1/2.0

Out[10]: 0.5

In [11]: 15%10

Out[11]: 5
```

Floor division is nothing but converting the result so obtained to the nearest integer.

```
In [12]: 2.8//2.0

Out[12]: 1.0
```

4.2 Relational Operators

Symbol	Task Performed
==	True, if it is equal
!=	True, if not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

```
In [13]: z = 1

In [14]: z == 1
```

```
Out[14]: True

In [15]: z > 1

Out[15]: False
```

4.3 Bitwise Operators

Symbol	Task Performed
&	Logical And
|	Logical OR
^	XOR
~	Negate
>>	Right shift
<<	Left shift

```
In [16]: a = 2 #10
         b = 3 #11

In [17]: print a & b
         print bin(a&b)

2
0b10

In [18]: 5 >> 1

Out[18]: 2
```

0000 0101 -> 5
Shifting the digits by 1 to the right and zero padding
0000 0010 -> 2

```
In [19]: 5 << 1

Out[19]: 10
```

0000 0101 -> 5
Shifting the digits by 1 to the left and zero padding
0000 1010 -> 10

5 Built-in Functions

Python comes loaded with pre-built functions

5.1 Conversion from one system to another

Conversion from hexadecimal to decimal is done by adding prefix **0x** to the hexadecimal value or vice versa by using built in **hex( )**, Octal to decimal by adding prefix **0** to the octal value or vice versa by using built in function **oct( )**.

```
In [20]: hex(170)

Out[20]: '0xaa'
```

```
In [21]: 0xAA

Out[21]: 170

In [22]: oct(8)

Out[22]: '010'

In [23]: 010

Out[23]: 8
```

int( ) accepts two values when used for conversion, one is the value in a different number system and the other is its base. Note that input number in the different number system should be of string type.

```
In [24]: print int('010',8)
         print int('0xaa',16)
         print int('1010',2)

8
170
10
```

int( ) can also be used to get only the integer value of a float number or can be used to convert a number which is of type string to integer format. Similarly, the function **str( )** can be used to convert the integer back to string format

```
In [25]: print int(7.7)
         print int('7')

7
7
```

Also note that function **bin( )** is used for binary and **float( )** for decimal/float values. **chr( )** is used for converting ASCII to its alphabet equivalent, **ord( )** is used for the other way round.

```
In [26]: chr(98)

Out[26]: 'b'

In [27]: ord('b')

Out[27]: 98
```

5.2 Simplifying Arithmetic Operations

round( ) function rounds the input value to a specified number of places or to the nearest integer.

```
In [28]: print round(5.6231)
         print round(4.55892, 2)

6.0
4.56
```

complex( ) is used to define a complex number and **abs( )** outputs the absolute value of the same.

```
In [29]: c =complex('5+2j')
         print abs(c)
```

```
5.38516480713
```

divmod(x,y) outputs the quotient and the remainder in a tuple(you will be learning about it in the further chapters) in the format (quotient, remainder).

```
In [30]: divmod(9,2)

Out[30]: (4, 1)
```

isinstance( ) returns True, if the first argument is an instance of that class. Multiple classes can also be checked at once.

```
In [31]: print isinstance(1, int)
         print isinstance(1.0,int)
         print isinstance(1.0,(int,float))

True
False
True
```

cmp(x,y)

x ? y	Output
x < y	-1
x == y	0
x > y	1

```
In [32]: print cmp(1,2)
         print cmp(2,1)
         print cmp(2,2)

-1
1
0
```

pow(x,y,z) can be used to find the power x^y also the mod of the resulting value with the third specified number can be found i.e. : (x^y % z).

```
In [33]: print pow(3,3)
         print pow(3,3,5)

27
2
```

range( ) function outputs the integers of the specified range. It can also be used to generate a series by specifying the difference between the two numbers within a particular range. The elements are returned in a list (will be discussing in detail later.)

```
In [34]: print range(3)
         print range(2,9)
         print range(2,27,8)

[0, 1, 2]
[2, 3, 4, 5, 6, 7, 8]
[2, 10, 18, 26]
```

5.3 Accepting User Inputs

raw_input( ) accepts input and stores it as a string. Hence, if the user inputs a integer, the code should convert the string to an integer and then proceed.

```
In [35]: abc = raw_input("Type something here and it will be stored in variable abc \t")

Type something here and it will be stored in variable abc         Hey

In [36]: type(abc)

Out[36]: str
```

input( ), this is used only for accepting only integer inputs.

```
In [37]: abc1 =  input("Only integer can be stored in in variable abc \t")

Only integer can be stored in in variable abc         275

In [38]: type(abc1)

Out[38]: int
```

Note that **type( )** returns the format or the type of a variable or a number

6 Print Statement

The **print** statement can be used in the following different ways :

```
- print "Hello World"
- print "Hello", <Variable Containing the String>
- print "Hello" + <Variable Containing the String>
- print "Hello %s" % <variable containing the string>
```

```
In [1]: print "Hello World"
```

```
Hello World
```

In Python, single, double and triple quotes are used to denote a string. Most use single quotes when declaring a single character. Double quotes when declaring a line and triple quotes when declaring a paragraph/multiple lines.

```
In [2]: print 'Hey'
```

```
Hey
```

```
In [3]: print """My name is Rajath Kumar M.P.

        I love Python."""
```

```
My name is Rajath Kumar M.P.

I love Python.
```

Strings can be assigned to variable say *string1* and *string2* which can called when using the print statement.

```
In [4]: string1 = 'World'
        print 'Hello', string1

        string2 = '!'
        print 'Hello', string1, string2
```

```
Hello World
Hello World !
```

String concatenation is the "addition" of two strings. Observe that while concatenating there will be no space between the strings.

```
In [5]: print 'Hello' + string1 + string2
```

```
HelloWorld!
```

%s is used to refer to a variable which contains a string.

```
In [6]: print "Hello %s" % string1
```

```
Hello World
```

Similarly, when using other data types

- %s -> string
- %d -> Integer
- %f -> Float
- %o -> Octal
- %x -> Hexadecimal
- %e -> exponential

This can be used for conversions inside the print statement itself.

```
In [7]: print "Actual Number = %d" %18
        print "Float of the number = %f" %18
        print "Octal equivalent of the number = %o" %18
        print "Hexadecimal equivalent of the number = %x" %18
        print "Exponential equivalent of the number = %e" %18
```

```
Actual Number = 18
Float of the number = 18.000000
Octal equivalent of the number = 22
Hexadecimal equivalent of the number = 12
Exponential equivalent of the number = 1.800000e+01
```

When referring to multiple variables parenthesis is used.

```
In [8]: print "Hello %s %s" %(string1,string2)
```

```
Hello World !
```

6.1 Other Examples

The following are other different ways the print statement can be put to use.

```
In [9]: print "I want %%d to be printed %s" %'here'
```

```
I want %d to be printed here
```

```
In [10]: print '_A'*10
```

```
_A_A_A_A_A_A_A_A_A_A
```

```
In [11]: print "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
```

```
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
```

```
In [12]: print "I want \\n to be printed."
```

```
I want \n to be printed.
```

```
In [13]: print """
         Routine:
         \t- Eat
         \t- Sleep\n\t- Repeat
         """
```

```
Routine:
        - Eat
        - Sleep
        - Repeat
```

7 PrecisionWidth and FieldWidth

Fieldwidth is the width of the entire number and precision is the width towards the right. One can alter these widths based on the requirements.

The default Precision Width is set to 6.

```
In [14]: "%f" % 3.121312312312

Out[14]: '3.121312'
```

Notice upto 6 decimal points are returned. To specify the number of decimal points, '%(field-width).(precisionwidth)f' is used.

```
In [15]: "%.5f" % 3.121312312312

Out[15]: '3.12131'
```

If the field width is set more than the necessary than the data right aligns itself to adjust to the specified values.

```
In [16]: "%9.5f" % 3.121312312312

Out[16]: '  3.12131'
```

Zero padding is done by adding a 0 at the start of fieldwidth.

```
In [17]: "%020.5f" % 3.121312312312

Out[17]: '00000000000003.12131'
```

For proper alignment, a space can be left blank in the field width so that when a negative number is used, proper alignment is maintained.

```
In [18]: print "% 9f" % 3.121312312312
         print "% 9f" % -3.121312312312

3.121312
-3.121312
```

'+' sign can be returned at the beginning of a positive number by adding a + sign at the beginning of the field width.

```
In [19]: print "%+9f" % 3.121312312312
         print "% 9f" % -3.121312312312

+3.121312
-3.121312
```

As mentioned above, the data right aligns itself when the field width mentioned is larger than the actualy field width. But left alignment can be done by specifying a negative symbol in the field width.

```
In [20]: "%-9.3f" % 3.121312312312

Out[20]: '3.121    '
```

8 Data Structures

In simple terms, It is the the collection or group of data in a particular structure.

8.1 Lists

Lists are the most commonly used data structure. Think of it as a sequence of data that is enclosed in square brackets and data are separated by a comma. Each of these data can be accessed by calling it's index value.

Lists are declared by just equating a variable to '[ ]' or list.

```
In [1]: a = []

In [2]: print type(a)

<type 'list'>
```

One can directly assign the sequence of data to a list x as shown.

```
In [3]: x = ['apple', 'orange']
```

8.1.1 Indexing

In python, Indexing starts from 0. Thus now the list x, which has two elements will have apple at 0 index and orange at 1 index.

```
In [4]: x[0]

Out[4]: 'apple'
```

Indexing can also be done in reverse order. That is the last element can be accessed first. Here, indexing starts from -1. Thus index value -1 will be orange and index -2 will be apple.

```
In [5]: x[-1]

Out[5]: 'orange'
```

As you might have already guessed, x[0] = x[-2], x[1] = x[-1]. This concept can be extended towards lists with more many elements.

```
In [6]: y = ['carrot','potato']
```

Here we have declared two lists x and y each containing its own data. Now, these two lists can again be put into another list say z which will have it's data as two lists. This list inside a list is called as nested lists and is how an array would be declared which we will see later.

```
In [7]: z  = [x,y]
        print z

[['apple', 'orange'], ['carrot', 'potato']]
```

Indexing in nested lists can be quite confusing if you do not understand how indexing works in python. So let us break it down and then arrive at a conclusion.

Let us access the data 'apple' in the above nested list. First, at index 0 there is a list ['apple','orange'] and at index 1 there is another list ['carrot','potato']. Hence z[0] should give us the first list which contains 'apple'.

```
In [8]: z1 = z[0]
        print z1
```

```
['apple', 'orange']
```

Now observe that z1 is not at all a nested list thus to access 'apple', z1 should be indexed at 0.

```
In [9]: z1[0]

Out[9]: 'apple'
```

Instead of doing the above, In python, you can access 'apple' by just writing the index values each time side by side.

```
In [10]: z[0][0]

Out[10]: 'apple'
```

If there was a list inside a list inside a list then you can access the innermost value by executing z[ ][ ][ ].

8.1.2 Slicing

Indexing was only limited to accessing a single element, Slicing on the other hand is accessing a sequence of data inside the list. In other words "slicing" the list.

Slicing is done by defining the index values of the first element and the last element from the parent list that is required in the sliced list. It is written as parentlist[a : b] where a,b are the index values from the parent list. If a or b is not defined then the index value is considered to be the first value for a if a is not defined and the last value for b when b is not defined.

```
In [11]: num = [0,1,2,3,4,5,6,7,8,9]

In [12]: print num[0:4]
         print num[4:]

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

You can also slice a parent list with a fixed length or step length.

```
In [13]: num[:9:3]

Out[13]: [0, 3, 6]
```

8.1.3 Built in List Functions

To find the length of the list or the number of elements in a list, **len( )** is used.

```
In [14]: len(num)

Out[14]: 10
```

If the list consists of all integer elements then **min( )** and **max( )** gives the minimum and maximum value in the list.

```
In [15]: min(num)

Out[15]: 0

In [16]: max(num)

Out[16]: 9
```

Lists can be concatenated by adding, '+' them. The resultant list will contain all the elements of the lists that were added. The resultant list will not be a nested list.

```
In [17]: [1,2,3] + [5,4,7]

Out[17]: [1, 2, 3, 5, 4, 7]
```

There might arise a requirement where you might need to check if a particular element is there in a predefined list. Consider the below list.

```
In [18]: names = ['Earth','Air','Fire','Water']
```

To check if 'Fire' and 'Rajath' is present in the list names. A conventional approach would be to use a for loop and iterate over the list and use the if condition. But in python you can use 'a in b' concept which would return 'True' if a is present in b and 'False' if not.

```
In [19]: 'Fire' in names

Out[19]: True

In [20]: 'Rajath' in names

Out[20]: False
```

In a list with elements as string, **max( )** and **min( )** is applicable. **max( )** would return a string element whose ASCII value is the highest and the lowest when **min( )** is used. Note that only the first index of each element is considered each time and if they value is the same then second index considered so on and so forth.

```
In [21]: mlist = ['bzaa','ds','nc','az','z','klm']

In [22]: print max(mlist)
         print min(mlist)

z
az
```

Here the first index of each element is considered and thus z has the highest ASCII value thus it is returned and minimum ASCII is a. But what if numbers are declared as strings?

```
In [23]: nlist = ['1','94','93','1000']

In [24]: print max(nlist)
         print min(nlist)

94
1
```

Even if the numbers are declared in a string the first index of each element is considered and the maximum and minimum values are returned accordingly.

But if you want to find the **max( )** string element based on the length of the string then another parameter 'key=len' is declared inside the **max( )** and **min( )** function.

```
In [25]: print max(names, key=len)
         print min(names, key=len)

Earth
Air
```

But even 'Water' has length 5. **max()** or **min()** function returns the first element when there are two or more elements with the same length.

Any other built in function can be used or lambda function (will be discussed later) in place of len.

A string can be converted into a list by using the **list()** function.

```
In [26]: list('hello')

Out[26]: ['h', 'e', 'l', 'l', 'o']
```

append( ) is used to add a element at the end of the list.

```
In [27]: lst = [1,1,4,8,7]

In [28]: lst.append(1)
         print lst

[1, 1, 4, 8, 7, 1]
```

count( ) is used to count the number of a particular element that is present in the list.

```
In [29]: lst.count(1)

Out[29]: 3
```

append( ) function can also be used to add a entire list at the end. Observe that the resultant list becomes a nested list.

```
In [30]: lst1 = [5,4,2,8]

In [31]: lst.append(lst1)
         print lst

[1, 1, 4, 8, 7, 1, [5, 4, 2, 8]]
```

But if nested list is not what is desired then **extend( )** function can be used.

```
In [32]: lst.extend(lst1)
         print lst

[1, 1, 4, 8, 7, 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

index( ) is used to find the index value of a particular element. Note that if there are multiple elements of the same value then the first index value of that element is returned.

```
In [33]: lst.index(1)

Out[33]: 0
```

insert(x,y) is used to insert a element y at a specified index value x. **append( )** function made it only possible to insert at the end.

```
In [34]: lst.insert(5, 'name')
         print lst

[1, 1, 4, 8, 7, 'name', 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

insert(x,y) inserts but does not replace element. If you want to replace the element with another element you simply assign the value to that particular index.

```
In [35]: lst[5] = 'Python'
         print lst
```

```
[1, 1, 4, 8, 7, 'Python', 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

pop( ) function return the last element in the list. This is similar to the operation of a stack. Hence it wouldn't be wrong to tell that lists can be used as a stack.

```
In [36]: lst.pop()
```

```
Out[36]: 8
```

Index value can be specified to pop a ceratin element corresponding to that index value.

```
In [37]: lst.pop(0)
```

```
Out[37]: 1
```

pop( ) is used to remove element based on it's index value which can be assigned to a variable. One can also remove element by specifying the element itself using the **remove( )** function.

```
In [38]: lst.remove('Python')
         print lst
```

```
[1, 4, 8, 7, 1, [5, 4, 2, 8], 5, 4, 2]
```

Alternative to **remove** function but with using index value is **del**

```
In [39]: del lst[1]
         print lst
```

```
[1, 8, 7, 1, [5, 4, 2, 8], 5, 4, 2]
```

The entire elements present in the list can be reversed by using the **reverse()** function.

```
In [40]: lst.reverse()
         print lst
```

```
[2, 4, 5, [5, 4, 2, 8], 1, 7, 8, 1]
```

Note that the nested list [5,4,2,8] is treated as a single element of the parent list lst. Thus the elements inside the nested list is not reversed.

Python offers built in operation **sort( )** to arrange the elements in ascending order.

```
In [41]: lst.sort()
         print lst
```

```
[1, 1, 2, 4, 5, 7, 8, [5, 4, 2, 8]]
```

For descending order, By default the reverse condition will be False for reverse. Hence changing it to True would arrange the elements in descending order.

```
In [42]: lst.sort(reverse=True)
         print lst
```

```
[[5, 4, 2, 8], 8, 7, 5, 4, 2, 1, 1]
```

Similarly for lists containing string elements, **sort( )** would sort the elements based on it's ASCII value in ascending and by specifying reverse=True in descending.

```
In [43]: names.sort()
         print names
         names.sort(reverse=True)
         print names

['Air', 'Earth', 'Fire', 'Water']
['Water', 'Fire', 'Earth', 'Air']
```

To sort based on length key=len should be specified as shown.

```
In [44]: names.sort(key=len)
         print names
         names.sort(key=len,reverse=True)
         print names

['Air', 'Fire', 'Water', 'Earth']
['Water', 'Earth', 'Fire', 'Air']
```

8.1.4 Copying a list

Most of the new python programmers commit this mistake. Consider the following,

```
In [45]: lista= [2,1,4,3]

In [46]: listb = lista
         print listb

[2, 1, 4, 3]
```

Here, We have declared a list, lista = [2,1,4,3]. This list is copied to listb by assigning it's value and it get's copied as seen. Now we perform some random operations on lista.

```
In [47]: lista.pop()
         print lista
         lista.append(9)
         print lista

[2, 1, 4]
[2, 1, 4, 9]

In [48]: print listb

[2, 1, 4, 9]
```

listb has also changed though no operation has been performed on it. This is because you have assigned the same memory space of lista to listb. So how do fix this?

If you recall, in slicing we had seen that parentlist[a:b] returns a list from parent list with start index a and end index b and if a and b is not mentioned then by default it considers the first and last element. We use the same concept here. By doing so, we are assigning the data of lista to listb as a variable.

```
In [49]: lista = [2,1,4,3]

In [50]: listb = lista[:]
         print listb

[2, 1, 4, 3]
```

```
In [51]: lista.pop()
         print lista
         lista.append(9)
         print lista
```

```
[2, 1, 4]
[2, 1, 4, 9]
```

```
In [52]: print listb
```

```
[2, 1, 4, 3]
```

8.2 Tuples

Tuples are similar to lists but only big difference is the elements inside a list can be changed but in tuple it cannot be changed. Think of tuples as something which has to be True for a particular something and cannot be True for no other values. For better understanding, Recall **divmod()** function.

```
In [53]: xyz = divmod(10,3)
         print xyz
         print type(xyz)
```

```
(3, 1)
<type 'tuple'>
```

Here the quotient has to be 3 and the remainder has to be 1. These values cannot be changed whatsoever when 10 is divided by 3. Hence divmod returns these values in a tuple.

To define a tuple, A variable is assigned to paranthesis ( ) or tuple( ).

```
In [54]: tup = ()
         tup2 = tuple()
```

If you want to directly declare a tuple it can be done by using a comma at the end of the data.

```
In [55]: 27,
```

```
Out[55]: (27,)
```

27 when multiplied by 2 yields 54, But when multiplied with a tuple the data is repeated twice.

```
In [56]: 2*(27,)
```

```
Out[56]: (27, 27)
```

Values can be assigned while declaring a tuple. It takes a list as input and converts it into a tuple or it takes a string and converts it into a tuple.

```
In [57]: tup3 = tuple([1,2,3])
         print tup3
         tup4 = tuple('Hello')
         print tup4
```

```
(1, 2, 3)
('H', 'e', 'l', 'l', 'o')
```

It follows the same indexing and slicing as Lists.

```
In [58]: print tup3[1]
         tup5 = tup4[:3]
         print tup5
```

```
2
('H', 'e', 'l')
```

8.2.1 Mapping one tuple to another

```
In [59]: (a,b,c)= ('alpha','beta','gamma')

In [60]: print a,b,c

alpha beta gamma

In [61]: d = tuple('RajathKumarMP')
         print d

('R', 'a', 'j', 'a', 't', 'h', 'K', 'u', 'm', 'a', 'r', 'M', 'P')
```

8.2.2 Built In Tuple functions

count() function counts the number of specified element that is present in the tuple.

```
In [62]: d.count('a')

Out[62]: 3
```

index() function returns the index of the specified element. If the elements are more than one then the index of the first element of that specified element is returned

```
In [63]: d.index('a')

Out[63]: 1
```

8.3 Sets

Sets are mainly used to eliminate repeated numbers in a sequence/list. It is also used to perform some standard set operations.

Sets are declared as set() which will initialize a empty set. Also set([sequence]) can be executed to declare a set with elements

```
In [64]: set1 = set()
         print type(set1)

<type 'set'>

In [65]: set0 = set([1,2,2,3,3,4])
         print set0

set([1, 2, 3, 4])
```

elements 2,3 which are repeated twice are seen only once. Thus in a set each element is distinct.

8.3.1 Built-in Functions

```
In [66]: set1 = set([1,2,3])

In [67]: set2 = set([2,3,4,5])
```

union( ) function returns a set which contains all the elements of both the sets without repition.

```
In [68]: set1.union(set2)

Out[68]: {1, 2, 3, 4, 5}
```

add( ) will add a particular element into the set. Note that the index of the newly added element is arbitrary and can be placed anywhere not neccessarily in the end.

```
In [69]: set1.add(0)
         set1

Out[69]: {0, 1, 2, 3}
```

intersection( ) function outputs a set which contains all the elements that are in both sets.

```
In [70]: set1.intersection(set2)

Out[70]: {2, 3}
```

difference( ) function ouptuts a set which contains elements that are in set1 and not in set2.

```
In [71]: set1.difference(set2)

Out[71]: {0, 1}
```

symmetric_difference( ) function ouputs a function which contains elements that are in one of the sets.

```
In [72]: set2.symmetric_difference(set1)

Out[72]: {0, 1, 4, 5}
```

issubset( ), isdisjoint( ), issuperset( ) is used to check if the set1/set2 is a subset, disjoint or superset of set2/set1 respectively.

```
In [73]: set1.issubset(set2)

Out[73]: False

In [74]: set2.isdisjoint(set1)

Out[74]: False

In [75]: set2.issuperset(set1)

Out[75]: False
```

pop( ) is used to remove an arbitrary element in the set

```
In [76]: set1.pop()
         print set1

set([1, 2, 3])
```

remove( ) function deletes the specified element from the set.

```
In [77]: set1.remove(2)
         set1

Out[77]: {1, 3}
```

clear( ) is used to clear all the elements and make that set an empty set.

```
In [78]: set1.clear()
         set1

Out[78]: set()
```

8.4 Strings

Strings are ordered text based data which are represented by enclosing the same in single/double/triple quotes.

```
In [1]: String0 = 'Taj Mahal is beautiful'
        String1 = "Taj Mahal is beautiful"
        String2 = '''Taj Mahal
        is
        beautiful'''

In [2]: print String0 , type(String0)
        print String1, type(String1)
        print String2, type(String2)

Taj Mahal is beautiful <type 'str'>
Taj Mahal is beautiful <type 'str'>
Taj Mahal
is
beautiful <type 'str'>
```

String Indexing and Slicing are similar to Lists which was explained in detail earlier.

```
In [3]: print String0[4]
        print String0[4:]

M
Mahal is beautiful
```

8.4.1 Built-in Functions

find( ) function returns the index value of the given data that is to found in the string. If it is not found it returns **-1**. Remember to not confuse the returned -1 for reverse indexing value.

```
In [4]: print String0.find('al')
        print String0.find('am')

7
-1
```

The index value returned is the index of the first element in the input data.

```
In [5]: print String0[7]

a
```

One can also input **find( )** function between which index values it has to search.

```
In [6]: print String0.find('j',1)
        print String0.find('j',1,3)

2
2
```

capitalize( ) is used to capitalize the first element in the string.

```
In [7]: String3 = 'observe the first letter in this sentence.'
        print String3.capitalize()
```

```
Observe the first letter in this sentence.
```

center( ) is used to center align the string by specifying the field width.

```
In [8]: String0.center(70)

Out[8]: '                         Taj Mahal is beautiful                         '
```

One can also fill the left out spaces with any other character.

```
In [9]: String0.center(70,'-')

Out[9]: '-------------------------Taj Mahal is beautiful-------------------------'
```

zfill( ) is used for zero padding by specifying the field width.

```
In [10]: String0.zfill(30)

Out[10]: '00000000Taj Mahal is beautiful'
```

expandtabs( ) allows you to change the spacing of the tab character. '\t' which is by default set to 8 spaces.

```
In [11]: s = 'h\te\tl\tl\to'
         print s
         print s.expandtabs(1)
         print s.expandtabs()

h        e        l        l        o
h e l l o
h       e       l       l       o
```

index( ) works the same way as **find( )** function the only difference is find returns '-1' when the input element is not found in the string but **index( )** function throws a ValueError

```
In [12]: print String0.index('Taj')
         print String0.index('Mahal',0)
         print String0.index('Mahal',10,20)

0
4


         ---------------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)

         <ipython-input-12-6062e7a32deb> in <module>()
           1 print String0.index('Taj')
           2 print String0.index('Mahal',0)
     ----> 3 print String0.index('Mahal',10,20)


         ValueError: substring not found
```

endswith( ) function is used to check if the given string ends with the particular char which is given as input.

```
In [13]: print String0.endswith('y')

False
```

The start and stop index values can also be specified.

```
In [14]: print String0.endswith('l',0)
         print String0.endswith('M',0,5)

True
True
```

count( ) function counts the number of char in the given string. The start and the stop index can also be specified or left blank. (These are Implicit arguments which will be dealt in functions)

```
In [15]: print String0.count('a',0)
         print String0.count('a',5,10)

4
2
```

join( ) function is used add a char in between the elements of the input string.

```
In [16]: 'a'.join('*_-')

Out[16]: '*a_a-'
```

'*_-'is the input string and char 'a' is added in between each element
join( ) function can also be used to convert a list into a string.

```
In [17]: a = list(String0)
         print a
         b = ''.join(a)
         print b

['T', 'a', 'j', ' ', 'M', 'a', 'h', 'a', 'l', ' ', 'i', 's', ' ', 'b', 'e', 'a', 'u', 't', 'i', 'f', 'u
Taj Mahal is beautiful
```

Before converting it into a string **join( )** function can be used to insert any char in between the list elements.

```
In [18]: c = '/'.join(a)[18:]
         print c

/i/s/ /b/e/a/u/t/i/f/u/l
```

split( ) function is used to convert a string back to a list. Think of it as the opposite of the **join()** function.

```
In [19]: d = c.split('/')
         print d

[' ', 'i', 's', ' ', 'b', 'e', 'a', 'u', 't', 'i', 'f', 'u', 'l']
```

In **split( )** function one can also specify the number of times you want to split the string or the number of elements the new returned list should conatin. The number of elements is always one more than the specified number this is because it is split the number of times specified.

```
In [20]: e = c.split('/',3)
         print e
         print len(e)
```

```
[' ', 'i', 's', ' /b/e/a/u/t/i/f/u/l']
4
```

lower( ) converts any capital letter to small letter.

```
In [21]: print String0
         print String0.lower()
```

```
Taj Mahal is beautiful
taj mahal is beautiful
```

upper( ) converts any small letter to capital letter.

```
In [22]: String0.upper()
```

```
Out[22]: 'TAJ MAHAL IS BEAUTIFUL'
```

replace( ) function replaces the element with another element.

```
In [23]: String0.replace('Taj Mahal','Bengaluru')
```

```
Out[23]: 'Bengaluru is beautiful'
```

strip( ) function is used to delete elements from the right end and the left end which is not required.

```
In [24]: f = '    hello      '
```

If no char is specified then it will delete all the spaces that is present in the right and left hand side of the data.

```
In [25]: f.strip()
```

```
Out[25]: 'hello'
```

strip( ) function, when a char is specified then it deletes that char if it is present in the two ends of the specified string.

```
In [26]: f = '   ***----hello---*******     '
```

```
In [27]: f.strip('*')
```

```
Out[27]: '   ***----hello---*******     '
```

The asterisk had to be deleted but is not. This is because there is a space in both the right and left hand side. So in strip function. The characters need to be inputted in the specific order in which they are present.

```
In [28]: print f.strip(' *')
         print f.strip(' *-')
```

```
----hello---
hello
```

lstrip( ) and **rstrip( )** function have the same functionality as strip function but the only difference is **lstrip( )** deletes only towards the left side and **rstrip( )** towards the right.

```
In [29]: print f.lstrip(' *')
         print f.rstrip(' *')
```

```
----hello---*******
   ***----hello---
```

8.5 Dictionaries

Dictionaries are more used like a database because here you can index a particular sequence with your user defined string.

To define a dictionary, equate a variable to { } or dict()

```
In [30]: d0 = {}
         d1 = dict()
         print type(d0), type(d1)
```

```
<type 'dict'> <type 'dict'>
```

Dictionary works somewhat like a list but with an added capability of assigning it's own index style.

```
In [31]: d0['One'] = 1
         d0['OneTwo'] = 12
         print d0
```

```
{'OneTwo': 12, 'One': 1}
```

That is how a dictionary looks like. Now you are able to access '1' by the index value set at 'One'

```
In [32]: print d0['One']
```

```
1
```

Two lists which are related can be merged to form a dictionary.

```
In [33]: names = ['One', 'Two', 'Three', 'Four', 'Five']
         numbers = [1, 2, 3, 4, 5]
```

zip( ) function is used to combine two lists

```
In [34]: d2 = zip(names,numbers)
         print d2
```

```
[('One', 1), ('Two', 2), ('Three', 3), ('Four', 4), ('Five', 5)]
```

The two lists are combined to form a single list and each elements are clubbed with their respective elements from the other list inside a tuple. Tuples because that is what is assigned and the value should not change.

Further, To convert the above into a dictionary. **dict( )** function is used.

```
In [35]: a1 = dict(d2)
         print a1
```

```
{'Four': 4, 'Five': 5, 'Three': 3, 'Two': 2, 'One': 1}
```

8.5.1 Built-in Functions

clear( ) function is used to erase the entire database that was created.

```
In [36]: a1.clear()
         print a1
```

```
{}
```

Dictionary can also be built using loops.

```
In [37]: for i in range(len(names)):
             a1[names[i]] = numbers[i]
         print a1

{'Four': 4, 'Five': 5, 'Three': 3, 'Two': 2, 'One': 1}
```

values( ) function returns a list with all the assigned values in the dictionary.

```
In [38]: a1.values()

Out[38]: [4, 5, 3, 2, 1]
```

keys( ) function returns all the index or the keys to which contains the values that it was assigned to.

```
In [39]: a1.keys()

Out[39]: ['Four', 'Five', 'Three', 'Two', 'One']
```

items( ) is returns a list containing both the list but each element in the dictionary is inside a tuple. This is same as the result that was obtained when zip function was used.

```
In [40]: a1.items()

Out[40]: [('Four', 4), ('Five', 5), ('Three', 3), ('Two', 2), ('One', 1)]
```

pop( ) function is used to get the remove that particular element and this removed element can be assigned to a new variable. But remember only the value is stored and not the key. Because the is just a index value.

```
In [41]: a2 = a1.pop('Four')
         print a1
         print a2

{'Five': 5, 'Three': 3, 'Two': 2, 'One': 1}
4
```

9 Control Flow Statements

9.1 If

if some_condition:

```
algorithm
```

```
In [1]: x = 12
        if x >10:
            print "Hello"
```

```
Hello
```

9.2 If-else

if some_condition:

```
algorithm
```

else:

```
algorithm
```

```
In [2]: x = 12
        if x > 10:
            print "hello"
        else:
            print "world"
```

```
hello
```

9.3 if-elif

if some_condition:

```
algorithm
```

elif some_condition:

```
algorithm
```

else:

```
algorithm
```

```
In [3]: x = 10
        y = 12
        if x > y:
            print "x>y"
        elif x < y:
            print "x<y"
        else:
            print "x=y"
```

```
x<y
```

if statement inside a if statement or if-elif or if-else are called as nested if statements.

```
In [4]: x = 10
        y = 12
        if x > y:
            print "x>y"
        elif x < y:
            print "x<y"
            if x==10:
                print "x=10"
            else:
                print "invalid"
        else:
            print "x=y"
```

```
x<y
x=10
```

9.4 Loops

9.4.1 For

for variable in something:

algorithm

```
In [5]: for i in range(5):
            print i
```

```
0
1
2
3
4
```

In the above example, i iterates over the 0,1,2,3,4. Every time it takes each value and executes the algorithm inside the loop. It is also possible to iterate over a nested list illustrated below.

```
In [6]: list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
        for list1 in list_of_lists:
                print list1
```

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

A use case of a nested for loop in this case would be,

```
In [7]: list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
        for list1 in list_of_lists:
            for x in list1:
                print x
```

```
1
2
3
4
5
6
7
8
9
```

9.4.2 While

while some_condition:

algorithm

```
In [8]: i = 1
        while i < 3:
            print(i ** 2)
            i = i+1
        print('Bye')
```

```
1
4
Bye
```

9.5 Break

As the name says. It is used to break out of a loop when a condition becomes true when executing the loop.

```
In [9]: for i in range(100):
            print i
            if i>=7:
                break
```

```
0
1
2
3
4
5
6
7
```

9.6 Continue

This continues the rest of the loop. Sometimes when a condition is satisfied there are chances of the loop getting terminated. This can be avoided using continue statement.

```
In [10]: for i in range(10):
             if i>4:
                 print "The end."
                 continue
             elif i<7:
                 print i
```

```
0
1
2
3
4
The end.
The end.
The end.
The end.
The end.
```

9.7 List Comprehensions

Python makes it simple to generate a required list with a single line of code using list comprehensions. For example If i need to generate multiples of say 27 I write the code using for loop as,

```
In [11]: res = []
         for i in range(1,11):
             x = 27*i
             res.append(x)
         print res
```

```
[27, 54, 81, 108, 135, 162, 189, 216, 243, 270]
```

Since you are generating another list altogether and that is what is required, List comprehensions is a more efficient way to solve this problem.

```
In [12]: [27*x for x in range(1,11)]
```

```
Out[12]: [27, 54, 81, 108, 135, 162, 189, 216, 243, 270]
```

That's it!. Only remember to enclose it in square brackets

Understanding the code, The first bit of the code is always the algorithm and then leave a space and then write the necessary loop. But you might be wondering can nested loops be extended to list comprehensions? Yes you can.

```
In [13]: [27*x for x in range(1,20) if x<=10]
```

```
Out[13]: [27, 54, 81, 108, 135, 162, 189, 216, 243, 270]
```

Let me add one more loop to make you understand better,

```
In [14]: [27*z for i in range(50) if i==27 for z in range(1,11)]
```

```
Out[14]: [27, 54, 81, 108, 135, 162, 189, 216, 243, 270]
```

10 Functions

Most of the times, In a algorithm the statements keep repeating and it will be a tedious job to execute the same statements again and again and will consume a lot of memory and is not efficient. Enter Functions.

This is the basic syntax of a function

def funcname(arg1, arg2,... argN):

```
''' Document String'''

statements


return <value>
```

Read the above syntax as, A function by name "funcname" is defined, which accepts arguements "arg1,arg2,....argN". The function is documented and it is '"Document String"'. The function after executing the statements returns a "value".

```
In [1]: print "Hey Rajath!"
        print "Rajath, How do you do?"
```

```
Hey Rajath!
Rajath, How do you do?
```

Instead of writing the above two statements every single time it can be replaced by defining a function which would do the job in just one line.

Defining a function firstfunc().

```
In [2]: def firstfunc():
            print "Hey Rajath!"
            print "Rajath, How do you do?"
```

```
In [3]: firstfunc()
```

```
Hey Rajath!
Rajath, How do you do?
```

firstfunc() every time just prints the message to a single person. We can make our function **firstfunc()** to accept arguements which will store the name and then prints respective to that accepted name. To do so, add a argument within the function as shown.

```
In [4]: def firstfunc(username):
            print "Hey", username + '!'
            print username + ',' ,"How do you do?"
```

```
In [5]: name1 = raw_input('Please enter your name : ')
```

```
Please enter your name : Guido
```

The name "Guido" is actually stored in name1. So we pass this variable to the function **firstfunc()** as the variable username because that is the variable that is defined for this function. i.e name1 is passed as username.

```
In [6]: firstfunc(name1)
```

```
Hey Guido!
Guido, How do you do?
```

Let us simplify this even further by defining another function **secondfunc()** which accepts the name and stores it inside a variable and then calls the **firstfunc()** from inside the function itself.

```
In [7]: def firstfunc(username):
            print "Hey", username + '!'
            print username + ',' ,"How do you do?"
        def secondfunc():
            name = raw_input("Please enter your name : ")
            firstfunc(name)

In [8]: secondfunc()

Please enter your name : karthik
Hey karthik!
karthik, How do you do?
```

10.1 Return Statement

When the function results in some value and that value has to be stored in a variable or needs to be sent back or returned for further operation to the main algorithm, return statement is used.

```
In [9]: def times(x,y):
            z = x*y
            return z
```

The above defined **times( )** function accepts two arguements and return the variable z which contains the result of the product of the two arguements

```
In [10]: c = times(4,5)
         print c

20
```

The z value is stored in variable c and can be used for further operations.

Instead of declaring another variable the entire statement itself can be used in the return statement as shown.

```
In [11]: def times(x,y):
             '''This multiplies the two input arguments'''
             return x*y

In [12]: c = times(4,5)
         print c

20
```

Since the **times( )** is now defined, we can document it as shown above. This document is returned whenever **times( )** function is called under **help( )** function.

```
In [13]: help(times)

Help on function times in module __main__:

times(x, y)
    This multiplies the two input arguments
```

Multiple variable can also be returned, But keep in mind the order.

```
In [14]: eglist = [10,50,30,12,6,8,100]

In [15]: def egfunc(eglist):
             highest = max(eglist)
             lowest = min(eglist)
             first = eglist[0]
             last = eglist[-1]
             return highest,lowest,first,last
```

If the function is just called without any variable for it to be assigned to, the result is returned inside a tuple. But if the variables are mentioned then the result is assigned to the variable in a particular order which is declared in the return statement.

```
In [16]: egfunc(eglist)

Out[16]: (100, 6, 10, 100)

In [17]: a,b,c,d = egfunc(eglist)
         print ' a =',a,'\n b =',b,'\n c =',c,'\n d =',d

a = 100
 b = 6
 c = 10
 d = 100
```

10.2 Implicit arguments

When an argument of a function is common in majority of the cases or it is "implicit" this concept is used.

```
In [18]: def implicitadd(x,y=3):
             return x+y
```

implicitadd( ) is a function accepts two arguments but most of the times the first argument needs to be added just by 3. Hence the second argument is assigned the value 3. Here the second argument is implicit.

Now if the second argument is not defined when calling the **implicitadd( )** function then it considered as 3.

```
In [19]: implicitadd(4)

Out[19]: 7
```

But if the second argument is specified then this value overrides the implicit value assigned to the argument

```
In [20]: implicitadd(4,4)

Out[20]: 8
```

10.3 Any number of arguments

If the number of arguments that is to be accepted by a function is not known then a asterisk symbol is used before the argument.

```
In [21]: def add_n(*args):
             res = 0
             reslist = []
             for i in args:
                 reslist.append(i)
             print reslist
             return sum(reslist)
```

The above function accepts any number of arguments, defines a list and appends all the arguments into that list and return the sum of all the arguments.

```
In [22]: add_n(1,2,3,4,5)
```

```
[1, 2, 3, 4, 5]
```

```
Out[22]: 15
```

```
In [23]: add_n(1,2,3)
```

```
[1, 2, 3]
```

```
Out[23]: 6
```

10.4 Global and Local Variables

Whatever variable is declared inside a function is local variable and outside the function in global variable.

```
In [24]: eg1 = [1,2,3,4,5]
```

In the below function we are appending a element to the declared list inside the function. eg2 variable declared inside the function is a local variable.

```
In [25]: def egfunc1():
             def thirdfunc(arg1):
                 eg2 = arg1[:]
                 eg2.append(6)
                 print "This is happening inside the function :", eg2
             print "This is happening before the function is called : ", eg1
             thirdfunc(eg1)
             print "This is happening outside the function :", eg1
             print "Accessing a variable declared inside the function from outside :" , eg2
```

```
In [26]: egfunc1()
```

```
This is happening before the function is called :  [1, 2, 3, 4, 5]
This is happening inside the function : [1, 2, 3, 4, 5, 6]
This is happening outside the function : [1, 2, 3, 4, 5]
Accessing a variable declared inside the function from outside :


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-26-949117e1ddc5> in <module>()
    ----> 1 egfunc1()


        <ipython-input-25-0da329480da9> in egfunc1()
          7     thirdfunc(eg1)
          8     print "This is happening outside the function :", eg1
    ----> 9     print "Accessing a variable declared inside the function from outside :" , eg2


        NameError: global name 'eg2' is not defined
```

If a **global** variable is defined as shown in the example below then that variable can be called from anywhere.

```
In [27]: eg3 = [1,2,3,4,5]
```

```
In [28]: def egfunc1():
             def thirdfunc(arg1):
                 global eg2
                 eg2 = arg1[:]
                 eg2.append(6)
                 print "This is happening inside the function :", eg2
             print "This is happening before the function is called : ", eg1
             thirdfunc(eg1)
             print "This is happening outside the function :", eg1
             print "Accessing a variable declared inside the function from outside :" , eg2
```

```
In [29]: egfunc1()
```

```
This is happening before the function is called :  [1, 2, 3, 4, 5]
This is happening inside the function : [1, 2, 3, 4, 5, 6]
This is happening outside the function : [1, 2, 3, 4, 5]
Accessing a variable declared inside the function from outside : [1, 2, 3, 4, 5, 6]
```

10.5 Lambda Functions

These are small functions which are not defined with any name and carry a single expression whose result is returned. Lambda functions comes very handy when operating with lists. These function are defined by the keyword **lambda** followed by the variables, a colon and the respective expression.

```
In [30]: z = lambda x: x * x
```

```
In [31]: z(8)
```

```
Out[31]: 64
```

10.5.1 map

map( ) function basically executes the function that is defined to each of the list's element separately.

```
In [32]: list1 = [1,2,3,4,5,6,7,8,9]
```

```
In [33]: eg = map(lambda x:x+2, list1)
         print eg
```

```
[3, 4, 5, 6, 7, 8, 9, 10, 11]
```

You can also add two lists.

```
In [34]: list2 = [9,8,7,6,5,4,3,2,1]
```

```
In [35]: eg2 = map(lambda x,y:x+y, list1,list2)
         print eg2
```

```
[10, 10, 10, 10, 10, 10, 10, 10, 10]
```

Not only lambda function but also other built in functions can also be used.

```
In [36]: eg3 = map(str,eg2)
         print eg3
```

```
['10', '10', '10', '10', '10', '10', '10', '10', '10']
```

10.5.2 filter

filter( ) function is used to filter out the values in a list. Note that **filter()** function returns the result in a new list.

```
In [37]: list1 = [1,2,3,4,5,6,7,8,9]
```

To get the elements which are less than 5,

```
In [38]: filter(lambda x:x<5,list1)
```

```
Out[38]: [1, 2, 3, 4]
```

Notice what happens when **map()** is used.

```
In [39]: map(lambda x:x<5, list1)
```

```
Out[39]: [True, True, True, True, False, False, False, False, False]
```

We can conclude that, whatever is returned true in **map( )** function that particular element is returned when **filter( )** function is used.

```
In [40]: filter(lambda x:x%4==0,list1)
```

```
Out[40]: [4, 8]
```

11 Classes

Variables, Lists, Dictionaries etc in python is a object. Without getting into the theory part of Object Oriented Programming, explanation of the concepts will be done along this tutorial.

A class is declared as follows

class class_name:

Functions

```
In [1]: class FirstClass:
            pass
```

pass in python means do nothing.

Above, a class object named "FirstClass" is declared now consider a "egclass" which has all the characteristics of "FirstClass". So all you have to do is, equate the "egclass" to "FirstClass". In python jargon this is called as creating an instance. "egclass" is the instance of "FirstClass"

```
In [2]: egclass = FirstClass()

In [3]: type(egclass)

Out[3]: instance

In [4]: type(FirstClass)

Out[4]: classobj
```

Now let us add some "functionality" to the class. So that our "FirstClass" is defined in a better way. A function inside a class is called as a "Method" of that class

Most of the classes will have a function named "__init__". These are called as magic methods. In this method you basically initialize the variables of that class or any other initial algorithms which is applicable to all methods is specified in this method. A variable inside a class is called an attribute.

These helps simplify the process of initializing a instance. For example,

Without the use of magic method or __init__ which is otherwise called as constructors. One had to define a **init( )** method and call the **init( )** function.

```
In [ ]: eg0 = FirstClass()
        eg0.init()
```

But when the constructor is defined the __init__ is called thus intializing the instance created.

We will make our "FirstClass" to accept two variables name and symbol.

I will be explaining about the "self" in a while.

```
In [6]: class FirstClass:
            def __init__(self,name,symbol):
                self.name = name
                self.symbol = symbol
```

Now that we have defined a function and added the __init__ method. We can create a instance of FirstClass which now accepts two arguments.

```
In [7]: eg1 = FirstClass('one',1)
        eg2 = FirstClass('two',2)

In [8]: print eg1.name, eg1.symbol
        print eg2.name, eg2.symbol
```

```
one 1
two 2
```

dir( ) function comes very handy in looking into what the class contains and what all method it offers

```
In [9]: dir(FirstClass)

Out[9]: ['__doc__', '__init__', '__module__']
```

dir( ) of an instance also shows it's defined attributes.

```
In [10]: dir(eg1)

Out[10]: ['__doc__', '__init__', '__module__', 'name', 'symbol']
```

Changing the FirstClass function a bit,

```
In [11]: class FirstClass:
             def __init__(self,name,symbol):
                 self.n = name
                 self.s = symbol
```

Changing self.name and self.symbol to self.n and self.s respectively will yield,

```
In [12]: eg1 = FirstClass('one',1)
         eg2 = FirstClass('two',2)

In [13]: print eg1.name, eg1.symbol
         print eg2.name, eg2.symbol


         ---------------------------------------------------------------------------

         AttributeError                            Traceback (most recent call last)

         <ipython-input-13-3717d682d1cf> in <module>()
     ----> 1 print eg1.name, eg1.symbol
           2 print eg2.name, eg2.symbol


         AttributeError: FirstClass instance has no attribute 'name'
```

AttributeError, Remember variables are nothing but attributes inside a class? So this means we have not given the correct attribute for the instance.

```
In [14]: dir(eg1)

Out[14]: ['__doc__', '__init__', '__module__', 'n', 's']

In [15]: print eg1.n, eg1.s
         print eg2.n, eg2.s

one 1
two 2
```

So now we have solved the error. Now let us compare the two examples that we saw.

When I declared self.name and self.symbol, there was no attribute error for eg1.name and eg1.symbol and when I declared self.n and self.s, there was no attribute error for eg1.n and eg1.s

From the above we can conclude that self is nothing but the instance itself.

Remember, self is not predefined it is userdefined. You can make use of anything you are comfortable with. But it has become a common practice to use self.

```
In [16]: class FirstClass:
             def __init__(asdf1234,name,symbol):
                 asdf1234.n = name
                 asdf1234.s = symbol

In [17]: eg1 = FirstClass('one',1)
         eg2 = FirstClass('two',2)

In [18]: print eg1.n, eg1.s
         print eg2.n, eg2.s

one 1
two 2
```

Since eg1 and eg2 are instances of FirstClass it need not necessarily be limited to FirstClass itself. It might extend itself by declaring other attributes without having the attribute to be declared inside the FirstClass.

```
In [19]: eg1.cube = 1
         eg2.cube = 8

In [20]: dir(eg1)

Out[20]: ['__doc__', '__init__', '__module__', 'cube', 'n', 's']
```

Just like global and local variables as we saw earlier, even classes have it's own types of variables.

Class Attribute : attributes defined outside the method and is applicable to all the instances.

Instance Attribute : attributes defined inside a method and is applicable to only that method and is unique to each instance.

```
In [21]: class FirstClass:
             test = 'test'
             def __init__(self,name,symbol):
                 self.name = name
                 self.symbol = symbol
```

Here test is a class attribute and name is a instance attribute.

```
In [22]: eg3 = FirstClass('Three',3)

In [23]: print eg3.test, eg3.name

test Three
```

Let us add some more methods to FirstClass.

```
In [24]: class FirstClass:
             def __init__(self,name,symbol):
                 self.name = name
                 self.symbol = symbol
             def square(self):
```

```
            return self.symbol * self.symbol
        def cube(self):
            return self.symbol * self.symbol * self.symbol
        def multiply(self, x):
            return self.symbol * x
```

```
In [25]: eg4 = FirstClass('Five',5)
```

```
In [26]: print eg4.square()
         print eg4.cube()
```

```
25
125
```

```
In [27]: eg4.multiply(2)
```

```
Out[27]: 10
```

The above can also be written as,

```
In [28]: FirstClass.multiply(eg4,2)
```

```
Out[28]: 10
```

11.1 Inheritance

There might be cases where a new class would have all the previous characteristics of an already defined class. So the new class can "inherit" the previous class and add it's own methods to it. This is called as inheritance.

Consider class SoftwareEngineer which has a method salary.

```
In [29]: class SoftwareEngineer:
             def __init__(self,name,age):
                 self.name = name
                 self.age = age
             def salary(self, value):
                 self.money = value
                 print self.name,"earns",self.money
```

```
In [30]: a = SoftwareEngineer('Kartik',26)
```

```
In [31]: a.salary(40000)
```

```
Kartik earns 40000
```

```
In [32]: dir(SoftwareEngineer)
```

```
Out[32]: ['__doc__', '__init__', '__module__', 'salary']
```

Now consider another class Artist which tells us about the amount of money an artist earns and his artform.

```
In [33]: class Artist:
             def __init__(self,name,age):
                 self.name = name
                 self.age = age
             def money(self,value):
                 self.money = value
                 print self.name,"earns",self.money
             def artform(self, job):
                 self.job = job
                 print self.name,"is a", self.job
```

```
In [34]: b = Artist('Nitin',20)

In [35]: b.money(50000)
         b.artform('Musician')

Nitin earns 50000
Nitin is a Musician

In [36]: dir(Artist)

Out[36]: ['__doc__', '__init__', '__module__', 'artform', 'money']
```

money method and salary method are the same. So we can generalize the method to salary and inherit the SoftwareEngineer class to Artist class. Now the artist class becomes,

```
In [37]: class Artist(SoftwareEngineer):
             def artform(self, job):
                 self.job = job
                 print self.name,"is a", self.job

In [38]: c = Artist('Nishanth',21)

In [39]: dir(Artist)

Out[39]: ['__doc__', '__init__', '__module__', 'artform', 'salary']

In [40]: c.salary(60000)
         c.artform('Dancer')

Nishanth earns 60000
Nishanth is a Dancer
```

Suppose say while inheriting a particular method is not suitable for the new class. One can override this method by defining again that method with the same name inside the new class.

```
In [41]: class Artist(SoftwareEngineer):
             def artform(self, job):
                 self.job = job
                 print self.name,"is a", self.job
             def salary(self, value):
                 self.money = value
                 print self.name,"earns",self.money
                 print "I am overriding the SoftwareEngineer class's salary method"

In [42]: c = Artist('Nishanth',21)

In [43]: c.salary(60000)
         c.artform('Dancer')

Nishanth earns 60000
I am overriding the SoftwareEngineer class's salary method
Nishanth is a Dancer
```

If not sure how many times methods will be called it will become difficult to declare so many variables to carry each result hence it is better to declare a list and append the result.

```
In [44]: class emptylist:
             def __init__(self):
                 self.data = []
             def one(self,x):
                 self.data.append(x)
             def two(self, x ):
                 self.data.append(x**2)
             def three(self, x):
                 self.data.append(x**3)

In [45]: xc = emptylist()

In [46]: xc.one(1)
         print xc.data

[1]
```

Since xc.data is a list direct list operations can also be performed.

```
In [47]: xc.data.append(8)
         print xc.data

[1, 8]

In [48]: xc.two(3)
         print xc.data

[1, 8, 9]
```

If the number of input arguments varies from instance to instance asterisk can be used as shown.

```
In [49]: class NotSure:
             def __init__(self, *args):
                 self.data = ''.join(list(args))

In [50]: yz = NotSure('I', 'Do' , 'Not', 'Know', 'What', 'To','Type')

In [51]: yz.data

Out[51]: 'IDoNotKnowWhatToType'
```

12 Where to go from here?

Practice. Give your self problem statements and solve them. You can also sign up to any competitive coding platform for problem statements. The more you code the more you discover and the more you start appreciating the language.

You can try out the different python libraries in the field of your interest. I highly recommend you to check out this curated list of Python frameworks, libraries and software http://awesome-python.com

The official python documentation : https://docs.python.org/2/

Peace.

Rajath Kumar M.P (rajathkumar.exe@gmail.com)