Basic Python:

1. Python(**Guido van Rossum**) is a high level language , easy to learn and it is an interpreted language not compiled language

**Source code** -> **intermediate binary code(.pyc)** which is a platform independent only versions matter-> **executed by an interpreted line by line** at run time like PHP which takes time on VM as we don’t mentioned any data type in python so it get executed line by line by interpreted

1. **Lambda function**: it is an anonymous function (defined without a name), while normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword for short period of time.

**lambda arguments: expression**

Lambda functions can have any number of arguments but only one expression

Ex: double = lambda x: x \* 2 # Output: 10

Print (double (5))

1. **Map , Reduce** and **Filter**:

r = map(func, seq)

The first argument *func* is the name of a function and the second a sequence (e.g. a list) *seq*. *map()* applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*

Ex :

>>> Celsius = [39.2, 36.5, 37.3, 37.8]

>>> Fahrenheit = map (lambda x: (float(9)/5)\*x + 32, Celsius)

Ex:

>>> a = [1,2,3,4]

>>> b = [17,12,11,10]

>>> c = [-1,-4,5,9]

>>> map(lambda x,y:x+y, a,b)

[18, 14, 14, 14]

>>> map(lambda x,y,z:x+y+z, a,b,c)

[17, 10, 19, 23]

**Filter:**

>>> result = filter(lambda x: x % 2 == 0, fib)

>>> print result

[0, 2, 8, 34]

This function will be applied to every element of the list *l*. Only if f returns True will the element of the list be included in the result list.

**Reduce:**

The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.

>>> reduce(lambda x,y: x+y, [47,11,42,13])

113

>>> reduce(lambda x, y: x+y, range(1,101))

5050

Shallow and Deep Copy:

From copy import **copy**

**Shallow copy:** In case of shallow copy, a reference of object is copied in other object. It means that any changes made to a copy of object do reflect in the original object.  
In python, this is implemented using “copy()” function.

lst1 = ['a','b',['ab','ba']]

lst2 = copy(lst1)

lst2[2][1] = "d"

lst2[0] = "c";

print lst2 ['c', 'b', ['ab', 'd']]

print lst1 ['c', 'b', ['ab', 'd']]

**Deep Copy:** In case of deep copy, a copy of object is copied in other object. It means that any changes made to a copy of object do not reflect in the original object.

From copy import **deepcopy**

lst1 = ['a','b',['ab','ba']]

lst2 = deepcopy(lst1)

lst2[2][1] = "d"

lst2[0] = "c";

print lst2 ['c', 'b', ['ab', 'd']]

print lst1['a', 'b', ['ab', 'ba']]

1. **Notes:**
2. Int, float, range, tuple and string are immutable (can’t change) in python

Ex:

**Immutable string:**

x = 'foo'

y = x

print x # foo

y += 'bar'

print x # foo

**Mutable List:**

x = [1, 2, 3]

y = x

print x # [1, 2, 3]

y += [3, 2, 1]

print x # [1, 2, 3, 3, 2, 1]

**Impo:** This can be avoided if we copy list with the slice operator as y = x[:] , In this case if we make changes in ‘y’ those won’t get effected in ‘x’. The changes will get affected in the original list(x) if there is a sublist as an element in the list and we modify it and that know as **shallow copy- we need to use deep copy to avoid such scenario**

1. As a list is mutable, it can't be used as a key in a dictionary, whereas a tuple can be used
2. Tuple method: count, to count the number of occurence of a value

**>>>** l = (1,2,3,1)

**>>>** l.count(1)

2

1. **List and generator comprehension:**

l = [1,2,3,4,5]

m = [print('Yes') if i == 1 else print ('No') for i in l]

n = (print('Yes') if i == 1 else print ('No') for i in l) , this will generate a generator object , we have to use list(n) to read the elements

1. Generators**:**

Generators are iterators, but you can only iterate over them once. It’s because they do not store all the values in memory, they generate the values on the fly:

* **Once the function yields, the function is paused and the control is transferred to the caller.**

Generators are a simple and powerful possibility to create or to generate iterators. On the surface they look like functions, but there is both a syntactical and a semantic difference. Instead of return statements you will find inside of the body of a generator only yield statements, i.e. one or more yield statements.   
  
Another important feature of generators is that the local variables and the execution start is automatically saved between calls. This is necessary, because unlike an ordinary function successive calls to a generator function don't start execution at the beginning of the function. Instead, the new call to a generator function will resume execution right after the yield statement in the code, where the last call exited. In other words: When the Python interpreter finds a yield statement inside of an iterator generated by a generator, it records the position of this statement and the local variables, and returns from the iterator. The next time this iterator is called, it will resume execution at the line following the previous yield statement.

**def** **yrange**(n):

i **=** 0

**while** i **<** n:

**yield** i

i **+=** 1

y **=** yrange(3)

y.next() #0

y.next() #1

y.next() #2

**Why we use them:**

Generators are good for calculating large sets of results (in particular calculations involving loops themselves) where you don't know if you are going to need all results, or where you don't want to allocate the memory for all results at the same time.

1. Decorators **:**

Python has an interesting feature called **decorators** to add functionality to an existing code. This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

**Basically, a decorator takes in a function, adds some functionality and returns it.**

1. Functions can be passed as arguments to another function.
2. a function can return another function

#### functions inside functions :

1. def f():
3. def g():
4. print("Hi, it's me 'g'")
5. print("Thanks for calling me")
7. print("This is the function 'f'")
8. print("I am calling 'g' now:")
9. g()
11. f()

#### Functions as parameter:

1. def g():
2. print("Hi, it's me 'g'")
3. print("Thanks for calling me")
5. def f(func):
6. print("Hi, it's me 'f'")
7. print("I will call 'func' now")
8. func()
10. f(g)

#### v) Functions returning Functions:

**def make\_cylinder\_volume\_func(r):**

**def volume (h):**

**return math.pi \* r \* r \* h**

**return volume**

**volume\_radius\_10 = make\_cylinder\_volume\_func (10)**

**volume\_radius\_10 (5)**

**=> 1570.7963267948967**

**vi) Decorator Example:**

**def smart\_divide(func):**

**def inner(a,b):**

**print("I am going to divide",a,"and",b)**

**if b == 0:**

**print("Whoops! cannot divide")**

**return**

**return func(a,b)**

**return inner**

**@smart\_divide**

**def divide(a,b):**

**return a/b**

**Output >>> divide (2, 5)**

**I am going to divide 2 and 5**

**0.4**

**Ex: 2)**

**import time**

**def timing\_function(some\_function):**

**def wrapper():**

**t1 = time.time()**

**some\_function()**

**t2 = time.time()**

**return "Time it took to run the function: " + str((t2 - t1)) + "\n"**

**return wrapper**

**@timing\_function**

**def my\_function():**

**num\_list = []**

**for num in (range(0, 10000)):**

**num\_list.append(num)**

**print("\nSum of all the numbers: " + str((sum(num\_list))))**

**print(my\_function()**

**Output:**

**>>**Sum of all the numbers: 49995000

>>Time it took to run the function: 0.05203509330749512

1. OOP in Python**:**
2. In Python, **instance variables** are variables whose value is assigned inside a constructor or method with self.

**Class variables** are variables whose value is assigned in class.

1. **Data hiding:** In Python, we use double underscore (Or \_\_) before the attributes name and those attributes will not be directly visible outside**.**
2. class MyClass:
4. # Hidden member of MyClass
5. \_\_hiddenVariable = 0
7. # A member method that changes
8. # \_\_hiddenVariable
9. def add(self, increment):
10. self.\_\_hiddenVariable += increment
11. print (self.\_\_hiddenVariable)
13. # Driver code
14. myObject = MyClass()
15. myObject.add(2)
16. myObject.add(5)
18. # This line causes error as we tried to access hidden variable outside the class using object and it threw an exception.
19. print (myObject.\_\_hiddenVariable)

**iii) Printing objects:** Printing objects gives us information about objects we are working with.

In python this can be achieved by using \_\_repr\_\_ or \_\_str\_\_ methods

**class Test:**

**def \_\_init\_\_(self, a, b):**

**self.a = a**

**self.b = b**

**def \_\_repr\_\_(self):**

**return "Test a:%s b:%s" % (self.a, self.b)**

**def \_\_str\_\_(self):**

**return "From str method of Test: a is %s," \**

**"b is %s" % (self.a, self.b)**

**# Driver Code**

**t = Test(1234, 5678)**

**print(t) # This calls \_\_str\_\_()**

**print([t]) # This calls \_\_repr\_\_()**

**Output :**

**From str method of Test: a is 1234,b is 5678**

**[Test a:1234 b:5678**]

Note:

1. **If no \_\_str\_\_ method is defined, print t (or print str(t)) uses \_\_repr\_\_.**
2. **If no \_\_repr\_\_ method is defined then the default is used.**

**iv) Inheritance:**

# Python example to show working of multiple

**# inheritance**

**class Base1(object):**

**def \_\_init\_\_(self):**

**self.str1 = "Geek1"**

**print "Base1"**

**class Base2(object):**

**def \_\_init\_\_(self):**

**self.str2 = "Geek2"**

**print "Base2"**

**class Derived(Base1, Base2):**

**def \_\_init\_\_(self):**

**# Calling constructors of Base1**

**# and Base2 classes**

**Base1.\_\_init\_\_(self)**

**Base2.\_\_init\_\_(self)**

**print "Derived"**

**def printStrs(self):**

**print(self.str1, self.str2)**

**ob = Derived()**

**ob.printStrs()**

**Output : Base1**

**Base2**

**Derived**

**('Geek1', 'Geek2')**

**v) How to check if a class is subclass of another?**

**print(issubclass(Derived, Base)) : True (need to pass the obj instances of the classes**)

**vi)** **How to access parent members in a subclass?**

Base class members can be accessed in derived class using base class name.

**vii) Super keyword:**

**# Python example to show that base**

**# class members can be accessed in**

**# derived class using super()**

**class Base(object):**

**# Constructor**

**def \_\_init\_\_(self, x):**

**self.x = x**

**class Derived(Base):**

**# Constructor**

**def \_\_init\_\_(self, x, y):**

**''' In Python 3.x, "super().\_\_init\_\_(name)"**

**also works'''**

**super(Derived, self).\_\_init\_\_(x)**

**self.y = y**

**def printXY(self):**

**# Note that Base.x won't work here**

**# because super() is used in constructor**

**print(self.x, self.y)**

**# Driver Code**

**d = Derived(10, 20)**

**d.printXY()**

1. **Sorting:**

k = [7,6,5,4,3,2,1]

##**In Order Sorting** (k will be updated)

k.sort()

print (k)

**Copied Sorting** (k will remain same the sorted contents will be copied to a new variable)

k1 = sorted(k)

print (k1)

print(k)

k = k1.sort(reverse = True)

print (k1)

1. Access modifiers in Python **(**Python does not have access modifiers. If you want to access an instance (or class) variable from outside the instance or class, you are always allowed to do so.):
2. All member variables and methods are **public** by default in Python. So when you want to make your member public, you just do nothing
3. **Protected** member is (in C++ and Java) accessible **only** from within the class and it’s subclasses. How to accomplish this in Python? The answer is – **by convention**. By prefixing the name of your member with **a single underscore**, you’re telling others “[don’t touch this, unless you’re a subclass](http://www.youtube.com/watch?v=otCpCn0l4Wo)”. See the example below

class Cup:

def \_\_init\_\_(self):

self.color = None

self.\_content = None

This changes virtually nothing, you’ll still be able to access the variable from outside the class using class name

1. By declaring your data member **private** you mean, that nobody should be able to access it from outside the class, i.e. strong [you can’t touch this](http://www.youtube.com/watch?v=otCpCn0l4Wo) policy.This feature turns every member name **prefixed with at least two underscores**

class Cup:

def \_\_init\_\_(self, color):

self.\_color = color # protected variable

self.\_\_content = None

**To access private parameter**:   
redCup **=** Cup("red")

redCup**.**\_Cup\_\_content **=** "tea"

1. Python 2 vs. Python 3 **:**

i) (print) and print

ii) Some libraries (like PyImage), only work on python 2.\* versions

iii) xrange(is a lazy evaluation it generates the xrange generator object- saves memory ) in python 2 and range in python 3 (xrange will give nameerror in python 3)

1. Integer Division : 3 / 2 = 1 in python 2 but 3 / 2 = 1.5 in python 3
2. Range vs. Xrange **:**

* **Xrange () of Python 2.x doesn’t exist in Python 3.x. In Python 2.x, range returns a list i.e. range(3) returns [0, 1, 2] while xrange returns a xrange object i. e., xrange(3) returns iterator object which work similar to Java iterator and generates number when needed.**
* **If we need to iterate over the same sequence multiple times, we prefer range () as range provides a static list. Xrange () reconstructs the sequence every time. Xrange () doesn’t support slices and other list methods. The advantage of xrange () is, it saves memory when task is to iterate over a large range.**
* **In Python 3.x, the range function now does what xrange does in Python 2.x, so to keep our code portable, we might want to stick to using range instead. So Python 3.x’s range function is xrange from Python 2.x.**
* **for x in xrange(1, 5):**

**print(x)**

* **for x in range(1, 5):**

**print(x),**

* 1. **Return type :**

**range() returns – the list as return type.**

**xrange() returns – xrange() object.**

* 1. **Memory :**

**The variable storing the range created by range () takes more memory as compared to variable storing the range using xrange (). The basic reason for this is the return type of range () is list and xrange () is xrange() object.**

* 1. **Operations Usage :**

**As range () returns the list, all the operations that can be applied on the list can be used on it. On the other hand, as xrange() returns the xrange object, operations associated to list cannot be applied on them, hence a disadvantage**

* 1. **Speed :**

**Because of the fact that xrange() evaluates only the generator object containing only the values that are required by lazy evaluation, therefore is faster in implementation than range().**

1. Numpy vs. python Array**:**

Python’s lists are efficient general-purpose containers. They support efficient for insertion, deletion, appending, and concatenation, and Python’s list comprehensions make them easy to construct and manipulate. However, they have certain limitations: they don’t support “**vectorized**” operations like element wise addition and multiplication, and the fact that they can contain objects of differing types mean that Python must store type information for every element, and must execute type dispatching code when operating on each element

1. **Is python a Call by Value or Call by Reference:**

In Python, by default, all the parameters (arguments) are passed “**by reference**” to the functions. Thus, if you change the value of the parameter within a function, the change is reflected in the calling function. We can even observe the pass “by value” kind of a behavior whenever we pass the arguments to functions that are of type say numbers, strings, and tuples. This is because of the immutable nature of them.

1. **Static (keyword, method, class) and class methods in python:**
   1. All variables defined on the class level in Python are considered static, we don’t need to use static keyword while declaring the variable as we do in java
   2. With static methods it gets a little more complex. In Python, there are two ways of defining static methods within a class 1) **@staticmethod 2) @classmethod**
   3. **@classmethod** :

i) A class method receives the class as implicit first argument, just like an instance method receives the instance.

ii) A class method is a method which is bound to the class and not the object of the class.

They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.

iii) It can modify a class state that would apply across all the instances of the class. For example it can modify a class variable that will be applicable to all the instances.

* 1. **@staticmethod :**
     1. A static method does not receive an implicit first argument.
     2. A static method is also a method which is bound to the class and not the object of the class.
     3. A static method can’t access or modify class state.
     4. It is present in a class because it makes sense for the method to be present in class.

**Example:**

**Implementation**

|  |
| --- |
| # Python program to demonstrate  # use of class method and static method.  from datetime import date    class Person:      def \_\_init\_\_(self, name, age):          self.name = name          self.age = age        # a class method to create a Person object by birth year.      @classmethod      def fromBirthYear(cls, name, year):          return cls(name, date.today().year - year)        # a static method to check if a Person is adult or not.      @staticmethod      def isAdult(age):          return age > 18    person1 = Person('mayank', 21)  person2 = Person.fromBirthYear('mayank', 1996)    print person1.age  print person2.age    # print the result  print Person.isAdult(22) |

1. **Inner Classes in Python:**

An inner class or nested class is a defined entirely within the body of another class. If an object is created using a class, the object inside the root class can be used. A class can have more than one inner classes, but in general inner classes are avoided.

class Human:

def \_\_init\_\_(self):

self.name = 'Guido'

self.head = self.Head()

self.brain = self.Brain()

class Head:

def talk(self):

return 'talking...'

class Brain:

def think(self):

return 'thinking...'

if \_\_name\_\_ == '\_\_main\_\_':

guido = Human()

print guido.name

print guido.head.talk()

print guido.brain.think()

* By using inner classes you can make your code even more object orientated. A single object can hold several sub objects.
* **(Advantage) Logical grouping of classes**: If a class is useful to only one other class then it is logical to embed it in that class and keep the two together
* We usually avoid using inner classes as it somehow effect the performance

1. **@property decorator in python :**

This is Python's way of creating getters, setters, and deleters (or [mutator methods](https://en.wikipedia.org/wiki/Mutator_method)) for a property in a class.

Python @property is one of the built-in decorators. The main purpose of any decorator is to change you class methods or attributes in such a way so that the user of your class no need to make any change in their code

Example:

1. **\*args and \*\*kwargs in python:**

1) The special syntax, \*args and \*\*kwargs in function definitions is used to

Pass a variable number of arguments to a function.

2) The single asterisk form (\*args) is used to pass a non-key worded,

Variable-length argument list

3) The double asterisk form is used to pass a key worded, variable-length

Argument list.

4) Kwargs prints the values in that form of key: value pair (Dictionary)

5) In kwargs we always use kwargs. Keys to get the keys value and for values use

Kwargs [key] and kwargs. Items() for the pairs

#args and kwargs in python

def method1(\*args,\*\*kwargs):

for arg in args:

print (arg)

for kwarg in kwargs.items():

print (kwarg)

#OR

for key in kwargs:

print (&#39;Pairs are: %s: %s&#39; % (key, kwargs[key]))

def main():

method1

(1, 2, 3, 4, w = 5, x = 11, y = 12, z = 13)

1. **Regular Expressions**
2. **MongoDb (MongoDb vs. SqlDb)**

**Note:**

1. **Numpy**, Scipy and Pandas are used for data wrangling and munging whereas Matplotlib for visualizing and to make sense of your data
2. Scikit Learn, Theano
3. Front-End Technology : HTML5,CSS3 and JavaScript
4. Hadoop
5. MongoDb
6. Crawling and Parsing of data in python