# OOPS in Python

**Python Object Oriented**

Python has been an object-oriented language

Here is small introduction of Object-Oriented Programming (OOP) to bring you at speed:

Overview of OOP Terminology

* **Class:** A user-defined prototype/blueprint for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
* **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables.
* **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
* **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects (arguments) involved.
* **Inheritance :** The transfer of the characteristics of a class to other classes that are derived from it.

**Child class can access the properties and methods of parent class**

* **Instance:** An individual object of a certain class..
* **Instantiation :** The creation of an instance of a class.
* **Method :** A special kind of function that is defined in a class definition.
* **Object :** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Creating Classes:

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows:

class ClassName:

'Optional class documentation string'

EXAMPLE:

Following is the example of a simple Python class:

* class Employee:  
   *'Common base class for all employees'* empCount = 0  
   def \_\_init\_\_(self, name, salary):  
   self.name = name  
   self.salary = salary  
   Employee.empCount += 1  
   def displayCount(self):  
   print ("Total Employee %d" % Employee.empCount)  
   def displayEmployee(self):  
   print ("Name : ", self.name,", Salary: ", self.salary)
* The variable *empCount* is a class variable whose value would be shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
* The first method *\_\_init\_\_()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
* You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you don't need to include it when you call the methods.

Creating instance objects:

To create instances of a class,

**you call the class using class name and pass in whatever arguments its*\_\_init\_\_* method accepts.**

"create first object of Employee class"  
emp1 = Employee("Testing", 2000)  
"create second object of Employee class"  
emp2 = Employee("World", 5000)

Accessing attributes:

emp1.displayEmployee()  
emp2.displayEmployee()  
print ("Total Employee %d" % Employee.empCount)

class Employee:  
 *'Common base class for all employees'* empCount = 0  
 def \_\_init\_\_(self, name, salary):  
 self.name = name  
 self.salary = salary  
 Employee.empCount += 1  
 def displayCount(self):  
 print ("Total Employee %d" % Employee.empCount)  
  
 def displayEmployee(self):  
 print ("Name : ", self.name,", Salary: ", self.salary)  
  
"create first object of Employee class"  
emp1 = Employee("Testing", 2000)  
"create second object of Employee class"  
emp2 = Employee("World", 5000)  
  
emp1.displayEmployee()  
emp2.displayEmployee()  
print ("Total Employee %d" % Employee.empCount)

You can add, remove or modify attributes of classes and objects at any time:

emp1.age = 7# Add an 'age' attribute.  
emp1.age = 8# Modify 'age' attribute.  
del emp1.age # Delete 'age' attribute.

Instead of using the normal statements to access attributes, you can use following functions:

* The **getattr(obj, name[, default])** : to access the attribute of object.
* The **hasattr(obj,name)** : to check if an attribute exists or not.
* The **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
* The **delattr(obj, name)** : to delete an attribute.
* hasattr(emp1, 'age')# Returns true if 'age' attribute exists  
  getattr(emp1, 'age')# Returns value of 'age' attribute  
  setattr(emp1, 'age', 8) # Set attribute 'age' at 8  
  delattr(emp1, 'age')# Delete attribute 'age'

Built-In Class Attributes:

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:

* **\_\_dict\_\_ :** Dictionary containing the class's namespace.
* **\_\_doc\_\_ :** Class documentation string or None if undefined.

Python deletes unneeded objects (built-in types or class instances) automatically to free memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed garbage collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it's assigned a new name or placed in a container (list, tuple or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

a = 40# Create object <40>

b = a # Increase ref. count of <40>

c = [b]# Increase ref. count of <40>

del a # Decrease ref. count of <40>

b = 100# Decrease ref. count of <40>

c[0] = -1# Decrease ref. count of <40>

You normally won't notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method *\_\_del\_\_()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any nonmemory resources used by an instance.

EXAMPLE:

class Point:  
 def \_\_init( self, x=0, y=0):  
 self.x = x  
 self.y = y  
 def \_\_del\_\_(self):  
 class\_name = self.\_\_class\_\_.\_\_name\_\_  
 print(class\_name, "destroyed")  
pt1 = Point()  
pt2 = pt1  
pt3 = pt1  
print (id(pt1), id(pt2), id(pt3)) # prints the ids of the obejcts  
del pt1  
del pt2  
#del pt3

**Note:** Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement.

# Class Inheritance:

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

SYNTAX:

Derived classes are declared much like their parent class; however, a list of base classes to inherit from are given after the class name:

class SubClassName (ParentClass1[, ParentClass2, ...]):

class Parent:# define parent class  
 parentAttr = 100  
 def \_\_init\_\_(self):  
 print ("Calling parent constructor")  
 def parentMethod(self):  
 print ('Calling parent method')  
 def setAttr(self, attr):  
 Parent.parentAttr = attr  
 def getAttr(self):  
 print ("Parent attribute :", Parent.parentAttr)  
class Child(Parent): # define child class  
 def \_\_init\_\_(self):  
 print ("Calling child constructor")  
 def childMethod(self):  
 print ('Calling child method')  
c = Child()# instance of child  
c.childMethod()# child calls its method  
c.parentMethod()# calls parent's method  
c.setAttr(200)# again call parent's method  
c.getAttr()# again call parent's method

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

Similar way, you can drive a class from multiple parent classes as follows:

class A:# define your class A

.....

class B:# define your calss B

.....

class C(A, B):# subclass of A and B

.....

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

* The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
* The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

**We have multiple and multilevel inheritences**

## Overriding Methods:

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

EXAMPLE:

from \_\_future\_\_ import print\_function  
class Parent:# define parent class  
 def myMethod(self):  
 print ('Calling parent method')  
class Child(Parent): # define child class  
 def myMethod(self):  
 print ('Calling child method')  
c = Child()# instance of child  
c.myMethod()# child calls overridden method

When the above code is executed, it produces the following result:

Calling child method