Python - Object Oriented

OOP Terminology

- **Class** − A user-defined prototype 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.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Operator overloading** − The assignment of more than one function to a particular operator.

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 −

Syntax :

```
class ClassName:
     'Optional class documentation string’
     class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Example :

```
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

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"
emp2 = Employee("Manni", 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 the 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.

Example :

```
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(empl, '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.
- __**name**__ − Class name.
- __**module**__ − Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- __**bases**__ − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Destroying Objects (Garbage Collection)

- Python deletes unneeded objects (built-in types or class instances) automatically to free the 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 is 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.

Example

```
a = 40          # Create object <40>
b = a           # Increase ref. count of <40>
C[0] = [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>
```

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
```

Class Inheritance

- Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
- 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 is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
     'Optional class documentation string’
     class_suite
```

Example

```
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
```

How to call a parent class constructor in Python

USE super().__init__() TO CALL THE IMMEDIATE PARENT CLASS CONSTRUCTOR

Call super().__init__(args) within the child class to call the constructor of the immediate parent class with the arguments args.

```
class Animal:
        def __init__(self, name):
                print("A " + name + " is an animal")

class Mammal(Animal):
        def __init__(self, name):
                super().__init__(name)
                print("A " + name + " is a mammal")

Mammal("Dog")
```

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

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

```
class A:    # define your class A
      .....

class B:    # define your class B
      .....

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

```
class Person:
    def __init__(self, name):
        print(name + " is a person")

class Athlete(Person):
    def __init__(self, name):
        print(name + " is an athlete")

class FamousPerson(Person):
    def __init__(self, name):
        print(name + " is a famous person")

class UsainBolt(Athlete, FamousPerson):
    def __init__(self):
        Athlete.__init__(self, "Usain Bolt")
        FamousPerson.__init__(self, "Usain Bolt")
        Person.__init__(self, "Usain Bolt")

UsainBolt()
```

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.

```
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
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

```
class JustCounter:
      __secretCount = 0

      def count(self):
            self.__secretCount += 1
            print self.__secretCount

counter = JustCounter()
counter.count() counter.count()
print counter.__secretCount
```