Object Oriented Programming in Python

Class, Object and Members)

**How to create an empty class**

# An empty class

class Test:

    pass

# A simple example class

class Test:

    # A sample method

    def fun(self):

        print("Hello")

# Driver code

#her simple object is created which belong to class “Test”

obj = Test()

obj.fun() # here method name “fun” is called

**The \_\_init\_\_ method**

The \_\_init\_\_ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

# A Sample class with init method

class Person:

    # init method or constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # Sample Method or behaviour

    def say\_hi(self):

        print('Hello, my name is', self.name)

#in order to create object of class “Person” you must provide no. of argument required in constructor of given class

p = Person('Shwetanshu')

p.say\_hi()

**Class and Instance Variables (Or attributes)**

In Python, instance variables are variables whose value is assigned inside a constructor or method with self.

# Class for Computer Science Student

class CSStudent:

    # Class Variable or class attributes

    stream = 'cse'

    # The init method or constructor

    def \_\_init\_\_(self, roll):

        # Instance Variable or object variable or instance attributes

        self.roll = roll

# Objects of CSStudent class

a = CSStudent(101)

b = CSStudent(102)

print(a.stream)  # prints "cse"

print(b.stream)  # prints "cse"

print(a.roll)    # prints 101

# Class variables can be accessed using classname also

print(CSStudent.stream) # prints "cse"

We can define instance variables inside normal methods also.

# Class for Computer Science Student

class CSStudent:

    # Class Variable

    stream = 'cse'

    # The init method or constructor

    def \_\_init\_\_(self, roll):

        # Instance Variable

        self.roll = roll

    # Adds an instance variable

    def setAddress(self, address):

        self.address = address

    # Retrieves instance variable

    def getAddress(self):

        return self.address

# Driver Code

# we can not use instance variable of any method without called it

a = CSStudent(101)

a.setAddress("Noida, UP")

print(a.getAddress())

# Inheritance in Python

Inheritance is the capability of one class to derive or inherit the properties from some another class. The benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides **reusability** of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.

# (Generally, object is made ancestor of all classes)

# In Python 3.x "class Person" is

# equivalent to "class Person(object)"

class Person(object):

    # Constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # To get name

    def getName(self):

        return self.name

    # To check if this person is employee

    def isEmployee(self):

        return False

# Inherited or Sub class (Note Person in bracket)

class Employee(Person):

    # Here we return true

    def isEmployee(self):

        return True

# Driver code

emp = Person("name1")  # An Object of Person

print(emp.getName(), emp.isEmployee())

emp = Employee("name2") # An Object of Employee

print(emp.getName(), emp.isEmployee())

# ****What is object class****

# **Subclassing (Calling constructor of parent class)** A child class needs to identify which class is its parent class. This can be done by mentioning the parent class name in the definition of the child class. Eg: class **subclass\_name (superclass\_name)**:

# Python code to demonstrate how parent constructors

# are called.

# parent class

class Person( object ):

        # \_\_init\_\_ is known as the constructor

        def \_\_init\_\_(self, name, idnumber):

                self.name = name

                self.idnumber = idnumber

        def display(self):

                print(self.name)

                print(self.idnumber)

# child class

class Employee( Person ):

        def \_\_init\_\_(self, name, idnumber, salary, post):

                self.salary = salary

                self.post = post

                # invoking the \_\_init\_\_ of the parent class

                Person.\_\_init\_\_(self, name, idnumber)

# creation of an object variable or an instance of Employee class

a = Employee('Rahul', 886012,20000,"assistant professor")

# accessing attributes of of the class Person using class Employee instance

print(a.name)

**Different forms of Inheritance:**  
**1. Single inheritance**: When a child class inherits from only one parent class, it is called as single inheritance. We saw an example above.

**2. Multiple inheritance**: When a child class inherits from multiple parent classes, it is called as multiple inheritance.  
Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as comma separated list in bracket.

# 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()

# ****Multilevel inheritance****: When we have child and grand child relationship.

# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.

# (Generally, object is made ancestor of all classes)

# In Python 3.x "class Person" is

# equivalent to "class Person(object)"

class Base(object):

    # Constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # To get name

    def getName(self):

        return self.name

# Inherited or Sub class (Note Person in bracket)

class Child(Base):

    # Constructor

    def \_\_init\_\_(self, name, age):

        Base.\_\_init\_\_(self, name)

        self.age = age

    # To get name

    def getAge(self):

        return self.age

# Inherited or Sub class (Note Person in bracket)

class GrandChild(Child):

    # Constructor

    def \_\_init\_\_(self, name, age, address):

        Child.\_\_init\_\_(self, name, age)

        self.address = address

    # To get address

    def getAddress(self):

        return self.address

# Driver code

g = GrandChild("Geek1", 23, "Noida")

print(g.getName(), g.getAge(), g.getAddress())

**4. Hierarchical inheritance** More than one derived classes are created from a single base.

**5. Hybrid inheritance**: This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

# Python program to demonstrate private members

# of the parent class

class C(object):

       def \_\_init\_\_(self):

              self.c = 21

              # d is private instance variable

              self.\_\_d = 42

class D(C):

       def \_\_init\_\_(self):

              self.e = 84

              C.\_\_init\_\_(self)

object1 = D()

# produces an error as d is private instance variable

print D.d

# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.

class Person(object):

    # Constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # To get name

    def getName(self):

        return self.name

    # To check if this person is employee

    def isEmployee(self):

        return False

# Inherited or Sub class (Note Person in bracket)

class Employee(Person):

    # Here we return true

    def isEmployee(self):

        return True

# Driver code

emp = Person("Geek1")  # An Object of Person

print(emp.getName(), emp.isEmployee())

emp = Employee("Geek2") # An Object of Employee

print(emp.getName(), emp.isEmployee())

Polymorphism and Overriding:

# Polymorphism means the ability to take various forms. In Python, Polymorphism allows us to define methods in the child class with the same name as defined in their parent class. As we know, a child class inherits all the methods from the parent class.

class Animal:

    def eating(self):

        print('Animal Eating!.')

class Lion(Animal):

    def eating(self):

        print('Lion is eating meat!.')

class Cow(Animal):

    def eating(self):

        print('Cow is eating grass!.')

#creating an object

a=Animal()

a.eating()

l=Lion()

l.eating()

c=Cow()

c.eating()

# Overriding:

# Whatever methods parent has by default available to the child through inheritance. Sometimes child may not satisfy with parent method implementation. Then child is allowed to redefine that method based on its requirement. This process is called overriding or overloading

# Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists.

# To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method \_\_add\_\_ is automatically invoked in which the operation for + operator is defined.

# There are 3 types of overloading:

# Operator Overloading

# Method Overloading

# Constructor Overloading

# Operator Overloading

class Point:

    def \_\_init\_\_(self, x = 0, y = 0):

        self.x = x

        self.y = y

    def \_\_str\_\_(self):

        return "({0},{1})".format(self.x,self.y)

    def \_\_add\_\_(self,other):

        x = self.x + other.x

        y = self.y + other.y

        return Point(x,y)

#creating first point object

p1=Point(10,20)

#creating second point object

p2=Point(30,40)

p3=Point(40,50)

p3=p1+p2+p3

print('Point1 =',p1)

print('Point2= ',p2)

print('Sum of Points= ',p3)

# 

# Method Overloading (Explain in polymorphism)

# Constructor Overloading

class Base(object):

    def \_\_init\_\_(self,a):

        self.a=a

        print("your are in base")

class Parents(Base):

    def \_\_init\_\_(self,b,c):

        print("you are in Parents class 1st constructor ")

    def ParentsM(self):

        print("parentsM")

    def \_\_init\_\_(self,d):

        print("your are in Parents class 2nd constructor")

class Child(Parents):

    def ChildM(self):

        print("you are in child class")

c=Child("hello")

c.ParentsM()

# Abstract Classes in Python (abstraction)

An abstract class can be considered as a blueprint for other classes, allows you to create a set of methods that must be created within any child classes built from your abstract class. A class which contains one or abstract methods is called an abstract class. An abstract method is a method that has declaration but not has any implementation. Abstract classes are not able to instantiated and it needs subclasses to provide implementations for those abstract methods which are defined in abstract classes. While we are designing large functional units we use an abstract class

from abc import ABC, abstractmethod

class Polygon(ABC):

    @abstractmethod

    def noofsides(self):

        pass

class Triangle(Polygon):

    # overriding abstract method

    def noofsides(self):

        print("I have 5 sides")

class Pentagon(Polygon):

    # overriding abstract method

    def noofsides(self):

        print("I have 5 sides")

class Hexagon(Polygon):

    # overriding abstract method

    def noofsides(self):

        print("I have 6 sides")

class Quadrilateral(Polygon):

    # overriding abstract method

    def noofsides(self):

        print("I have 4 sides")

# Driver code

R = Triangle()

R.noofsides()

K = Quadrilateral()

K.noofsides()

R = Pentagon()

R.noofsides()

K = Hexagon()

K.noofsides()