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

    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

    stream = 'cse'

    # The init method or constructor

    def \_\_init\_\_(self, roll):

        # Instance Variable or object variable

        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 class

# name 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())

## setattr() works in Python

class Person:

    name = 'Adam'# class variable

p = Person()

print('Before modification:', p.name)

# setting class variable "name"  from "Adam" to 'John'

setattr(p, 'name', 'John')

print('After modification:', p.name)

OUTPUT:

Before modification: Adam

After modification: John

## When the attribute is not found in setattr()

class Person:

    name = 'Adam'

p = Person()

# setting attribute name to John

setattr(p, 'name', 'John')

print('Name is:', p.name)

# IF setting an attribute not present in Person It will create it

setattr(p, 'age', 23)

print('Age is:', p.age)

OUTPUT:

Name is: John

Age is: 23

## getattr() works in Python

class Person:

    age = 23

    name = "Adam"

person = Person()

print('The age is:', getattr(person, "age"))

print('The age is:', person.age)

OUTPUT:

The age is: 23

The age is: 23

## getattr() when named attribute is not found

class Person:

    age = 23

    name = "Adam"

person = Person()

# when default value is provided

print('The sex is:', getattr(person, 'sex', 'Male'))

# when no default value is provided

print('The sex is:', getattr(person, 'sex'))

OUTPUT:

The sex is:Male

AttributeError: 'Person' object has no attribute 'sex'

## hasattr() works in Python

class Person:

    age = 23

    name = 'Adam'

person = Person()

print('Person has age?:', hasattr(person, 'age'))

print('Person has salary?:', hasattr(person, 'salary'))

OUTPUT:

Person has age?: True

Person has salary?: False

# (Data Hiding and Object Printing) / Encapsulation

**Data hiding**

In Python, we use double underscore (Or \_\_) before the attributes name and those attributes will not be directly visible outside.

class MyClass:

    # Hidden member of MyClass

    \_\_hiddenVariable = 0

    # A member method that changes

    # \_\_hiddenVariable

    def add(self, increment):

        self.\_\_hiddenVariable += increment

        print (self.\_\_hiddenVariable)

# Driver code

myObject = MyClass()

myObject.add(2)

myObject.add(5)

# This line causes error

print (myObject.\_\_hiddenVariable)

We can access the value of hidden attribute by a tricky syntax:

# A Python program to demonstrate that hidden

# members can be accessed outside a class

class MyClass:

    # Hidden member of MyClass

    \_\_hiddenVariable = 10

# Driver code

myObject = MyClass()

print(myObject.\_MyClass\_\_hiddenVariable)

**Printing Objects**

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\_\_()

**Important Points about Printing:**

If no \_\_str\_\_ method is defined, print t (or print str(t)) uses \_\_repr\_\_

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)

# Driver Code

t = Test(1234, 5678)

print(t)

If no \_\_repr\_\_ method is defined then the default is used.

class Test:

    def \_\_init\_\_(self, a, b):

        self.a = a

        self.b = b

# Driver Code

t = Test(1234, 5678)

print(t)

**Class Method: It is used to change the value of class variable though object/instance**

class Base:

    h1=30

    def \_\_init\_\_(self,a):

        self.a=a

        print("your are in base")

    @classmethod

    def changeh1(cls,h2): # where cls is act as Class name i.e Base

        cls.h1=h2

c=Base("hello")

c.changeh1(40)

print(c.h1)

**Class Method: It is used as constructor**

class Base:

    h1=30

    def \_\_init\_\_(self,name,age):

        self.name=name

        self.age=age

    @classmethod

    def change(r,string): # where r is act as Class name i.e Base

        element=string.split(",")

        return r(element[0],element[1])

a=Base.change("amir,29")

print(a.name)

**Static Method: It is used for creating normal function in class without using self or class as argument in function**

class Base:

    h1=30

    def \_\_init\_\_(self,name,age):

        self.name=name

        self.age=age

    @staticmethod

    def add(x,y):

        print(x+y)

a=Base("rahul",30)

a.add(5,6)

**use of super() : It is used for access class variable , instance variable(attributes), method of base class i.e parent class i.e super class**

class A(object):

    name="i am in class A variable"

    def \_\_init\_\_(self):

        self.name="i am in class A constructor"

class B(A):

    name="i am in class B variable"

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.name="i am in class B constructor"

a=B()

print(a.name)

**Accessing instance variable (attributes) of base Class A:**

class A(object):

    name="i am in class A variable"

    def \_\_init\_\_(self,age,role):

        self.name="i am in class A constructor"

        self.age=age

        self.role=role

class B(A):

    name="i am in class B variable"

    def \_\_init\_\_(self,age,role):

        super().\_\_init\_\_(age,role)

        self.name="i am in class B constructor"

a=B(25,"player")

print(a.role)

# 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("Geek1")  # An Object of Person

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

emp = Employee("Geek2") # 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())

**How to check if a class is subclass of another**

Python provides a function issubclass() that directly tells us if a class is subclass of another class.

# Python example to check if a class is

# subclass of another

class Base(object):

    pass   # Empty Class

class Derived(Base):

    pass   # Empty Class

# Driver Code

print(issubclass(Derived, Base))

print(issubclass(Base, Derived))

d = Derived()

b = Base()

# b is not an instance of Derived

print(isinstance(b, Derived))

# But d is an instance of Base

print(isinstance(d, Base))

**Multiple 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()

**How to access parent members(attributes ,methods) in a subclass**

1. **Using Parent class name**

# Python example to show that base

# class members can be accessed in

# derived class using base class name

class Base(object):

    # Constructor

    def \_\_init\_\_(self, x):

        self.x = x

class Derived(Base):

    # Constructor

    def \_\_init\_\_(self, x, y):

        Base.x = x

        self.y = y

    def printXY(self):

       # print(self.x, self.y) will also work

       print(Base.x, self.y)

# Driver Code

d = Derived(10, 20)

d.printXY()

1. **Using super()**

# 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()

**Constructor working in inheritence:**

# Python example  to check which constructor is working for creating any object

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

# Python program showing

# abstract base class work

from abc import ABC, abstractmethod

class Polygon(ABC):

    # abstract method

    def noofsides(self):

        pass

class Triangle(Polygon):

    # overriding abstract method

    def noofsides(self):

        print("I have 3 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()