**INTRODUCTION TO OOPS:**

* Python is an Object Oriented Programming language.
* Classes and Objects are the key features of OOPs.
* Class is the main building block and acts as template for the object.
* Object is a collection of data and functions and are also called as instances of a class or class variable.
* In Python, everything is treated as an object. For example, all integer variables that we use in our program is an object of class int. Similarly all string variables are also object of class string.

**DEFINING CLASSES:**

* Class is a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
* A class is like a blueprint for an object and is defined by using the keyword **class**.
* **Syntax:**

class class\_name:

statement\_1

statement\_2

…………..

…………..

Statement\_n

**LET’S LEARN SOMETHING NEW**

* Variables defined inside a class are called as “Class Variable” and functions are called as “Methods”.
* Class variable and methods are together known as members of the class.
* The class members should be accessed through objects or instance of class.
* **Example:** Program to define a class

**class Sample:**

**x, y = 10, 20 # class variables**

* In the above code, name of the class is Sample and it has two variables x and y having the initial value 10 and 20 respectively.
* To access the values defined inside the class, we need to create an object or instance of that class.

**CREATING OBJECTS:**

* Once a class is created, it’s members can be accessed by the object or instance of that class.
* The process of creating object is called as “**Class Instantiation**”.
* **Syntax:**

Object\_name = class\_name( )

* **Example:**

obj1 = Sample( )

* We are creating an object named obj1 for the class Sample.
* Any class member ie. class variable or method (function) can be accessed by using the object with a dot ( . ) operator.
* **Syntax**: Object\_name . class\_member
* **Example:** Program to define a class and access its member variables.

**class Sample:**

**x, y = 10, 20 #class variables**

**S=Sample( ) # class instantiation**

**print("Value of x = ", S.x)**

**print("Value of y = ", S.y)**

**print("Sum of x and y = ", S.x+S.y)**

**CLASS METHODS:**

* Class function or method is very similar to ordinary function with a small difference that, the class method must have the first argument named as **self**.
* No need to pass a value for this self argument as Python provides its value automatically.
* Even if a method takes no arguments, it should be defined with self.
* If a method is defined to accept only one argument it will take it as two arguments ie. self and the defined argument.
* When you declare class variable within class, methods must be prefixed with the class name and dot operator.

Example:

**Program to find total and average marks using class:**

class Student:

mark1, mark2, mark3 = 45, 91, 71 #class variable

def process(self): #class method

sum = Student.mark1 + Student.mark2 + Student.mark3

avg = sum/3

print("Total Marks = ", sum)

print("Average Marks = ", avg)

return

S=Student()

S.process()

Or

class student():

#English,Hindi,Science=88,90,100

English=int(input("Enter english marks"))

Hindi=int(input("Enter hindi marks"))

science=int(input("Enter science marks"))

def p(self,evs):

total=self.English+self.Hindi+self.science+evs

avg=total/3

print(avg)

S=student()

evs=90

S.p(evs)

print(S.English)

or

class sample:

def process(self,Marks1,Marks2,Marks3):

self.marks1=Marks1

self.marks2=Marks2

self.marks3=Marks3

sum= self.marks1+ self.marks2+self.marks3

avg= sum/3

print(avg)

Marks1=88

Marks2=88

Marks3=88

s=sample()

s.process(Marks1,Marks2,Marks3)

**CONSTRUCTORS:**

* Constructor is a special function that is automatically executed when an object of a class is created.
* “\_\_init\_\_” acts as a constructor in python.
* It must begin and end with double underscore. This function will act as an ordinary function; but only difference is, it is executed automatically when the object is created.
* This constructor function can be defined with or without arguments.
* It is used to initialize the class variables.

**General syntax of \_\_init\_\_ method (Constructor):**

def \_\_init\_\_(self, [args ……..]):

<statements>

**Example:** Program to illustrate Constructor

class Sample:

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

print("Constructor of class Sample...")

self.num=num

print("The value is :",self. num)

S=Sample(10)

**DESTRUCTORS:**

* Destructors are called when an object gets destroyed. It is the opposite to constructors.
* The \_\_[del](https://www.geeksforgeeks.org/delattr-del-python/)\_\_() method is known as destructor in Python.
* **Syntax of destructor declaration :**

def \_\_del\_\_(self):

# body of destructor

**class** Employee:

# Initializing

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

**print**('Employee created.')

# Deleting (Calling destructor)

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

print('Destructor called, Employee deleted.')

obj **=** Employee()

**del** obj

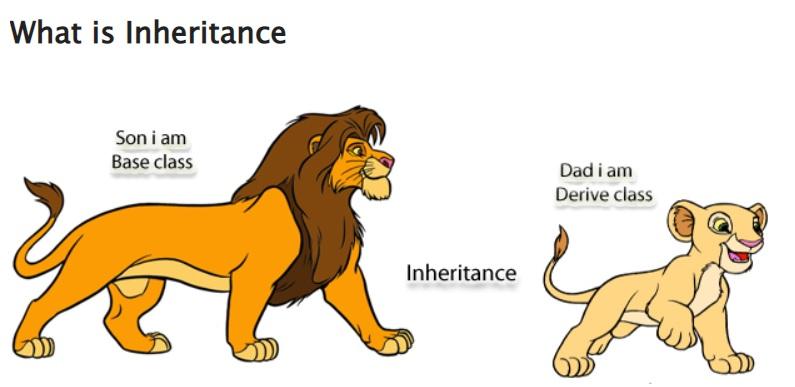
**Output**

Employee created.

Destructor called, Employee deleted.

**Inheritance:**

* Inheritance refers to the process of creating new classes from existing classes.
* Inheritance provides reusability of a code and allows us to add more features to a class without modifying it.
* The new class is called derived (or child) class
* The one from which it inherits is called the base (or parent) class.



The **four** main types of inheritance are :

1) Single Inheritance

2) Multilevel Inheritance

3) Hierarchical Inheritance

4) Multiple Inheritance

5)Hybrid

1. **SINGLE INHERITANCE:**

Single inheritance enables a derived class to inherit properties from a single parent class.

**Syntax:**

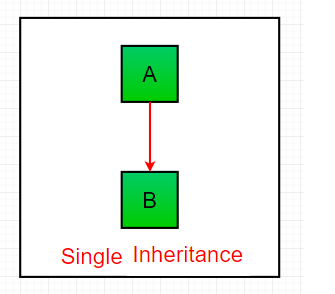
Class A:

# Properties of class A

Class B(A):

# Class B inheriting property of class A

# more properties of class B



**class** Parent:

**def** func1(self):

print("This function is in parent class.")

# Derived class

**class** Child(Parent):

**def** func2(self):

**print**("This function is in child class.")

# Driver's code

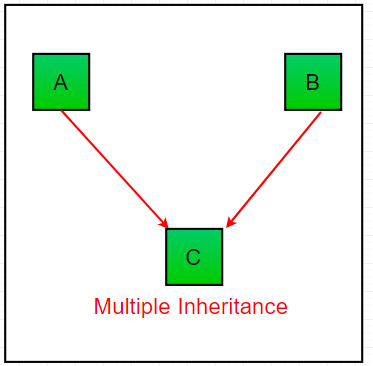
object **=** Child()

object.func1()

object.func2()

### **Multiple Inheritance:**

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



# Base class1

**class** Mother:

mothername **=** ""

**def** mother(self):

print(self.mothername)

# Base class2

**class** Father:

fathername **=** ""

**def** father(self):

**print**(self.fathername)

# Derived class

**class** Son(Mother, Father):

**def** parents(self):

print("Father :", self.fathername)

print("Mother :", self.mothername)

# Driver's code

s1 **=** Son()

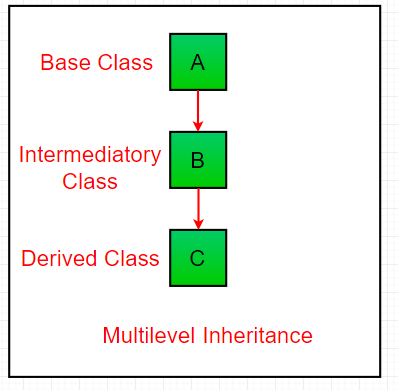
s1.fathername **=** "RAM"

s1.mothername **=** "SITA"

s1.parents()

### **Multilevel Inheritance :**

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



**class** Grandfather:

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

self.grandfathername **=** grandfathername

# Intermediate class

**class** Father(Grandfather):

**def** \_\_init\_\_(self, fathername, grandfathername):

self.fathername **=** fathername

# invoking constructor of Grandfather class

Grandfather.\_\_init\_\_(self, grandfathername)

# Derived class

**class** Son(Father):

**def** \_\_init\_\_(self, sonname, fathername, grandfathername):

self.sonname **=** sonname

# invoking constructor of Father class

Father.\_\_init\_\_(self, fathername, grandfathername)

**def** print\_name(self):

print('Grandfather name :', self.grandfathername)

**print**("Father name :", self.fathername)

**print**("Son name :", self.sonname)

# Driver code

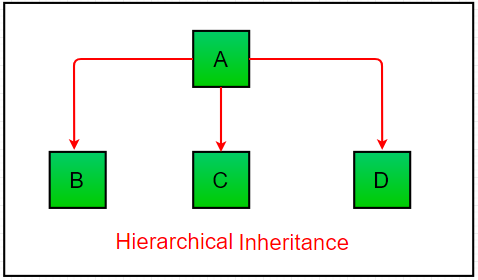
s1 **=** Son('Prince', 'Rampal', 'Lal mani')

**print**(s1.grandfathername)

s1.print\_name()

### **Hierarchical Inheritance:**

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and 3 child (derived) classes.



# Base class

**class** Parent:

**def** func1(self):

**print**("This function is in parent class.")

# Derived class1

**class** Child1(Parent):

**def** func2(self):

**print**("This function is in child 1.")

# Derivied class2

**class** Child2(Parent):

**def** func3(self):

**print**("This function is in child 2.")

# Driver's code

object1 **=** Child1()

object2 **=** Child2()

object1.func1()

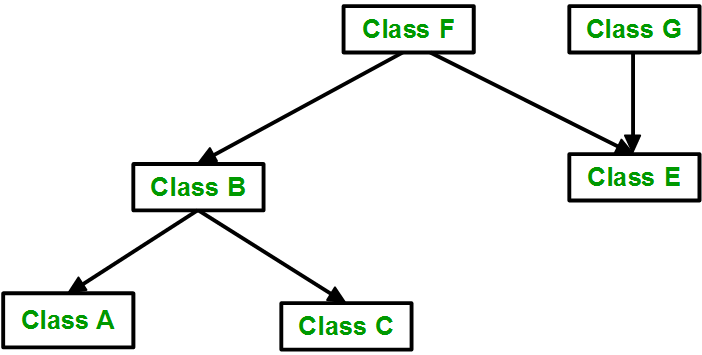
object1.func2()

object2.func1()

object2.func3()

### **Hybrid Inheritance:**

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.



**class** School:

**def** func1(self):

**print**("This function is in school.")

**class** Student1(School):

**def** func2(self):

**print**("This function is in student 1. ")

**class** Student2(School):

**def** func3(self):

**print**("This function is in student 2.")

**class** Student3(Student1, School):

**def** func4(self):

**print**("This function is in student 3.")

# Driver's code

object **=** Student3()

object.func1()

object.func2()

# **Polymorphism in Python**

**What is Polymorphism:** The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Example 1)# Python program to demonstrate in-built poly-

# morphic functions

# len() being used for a string

print(len("geeks"))

# len() being used for a list

**print**(len([10, 20, 30]))

Example 2)# A simple Python function to demonstrate

# Polymorphism

**def** add(x, y, z **=** 0):

**return** x **+** y**+**z

# Driver code

**print**(add(2, 3))

print(add(2, 3, 4))

Example 3)

**class** India():

**def** capital(self):

**print**("New Delhi is the capital of India.")

**def** language(self):

print("Hindi is the most widely spoken language of India.")

**def** type(self):

print("India is a developing country.")

**class** USA():

**def** capital(self):

print("Washington, D.C. is the capital of USA.")

**def** language(self):

print("English is the primary language of USA.")

**def** type(self):

print("USA is a developed country.")

obj\_ind **=** India()

obj\_usa **=** USA()

**for** country **in** (obj\_ind, obj\_usa):

country.capital()

country.language()

country.type()

# **Encapsulation in Python**

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object’s variable can only be changed by an object’s method. Those types of variables are known as **private variables.**

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc. The goal of information hiding is to ensure that an object’s state is always valid by controlling access to attributes that are hidden from the outside world.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when due to some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”. Using encapsulation also hides the data. In this example, the data of the sections like sales, finance, or accounts are hidden from any other section.

# Python program to

# demonstrate protected members

# Creating a base class

**class** Base:

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

# Protected member

self.\_a **=** 2

# Creating a derived class

**class** Derived(Base):

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

# Calling constructor of

# Base class

Base.\_\_init\_\_(self)

**print**("Calling protected member of base class: ",

self.\_a)

# Modify the protected variable:

self.\_a **=** 3

**print**("Calling modified protected member outside class: ",

self.\_a)

obj1 **=** Derived()

obj2 **=** Base()

# Calling protected member

# Can be accessed but should not be done due to convention

print("Accessing protected member of obj1: ", obj1.\_a)

# Accessing the protected variable outside

print("Accessing protected member of obj2: ", obj2.\_a)

# **Abstract Classes in Python**

An abstract class can be considered a blueprint for other [classes](https://www.geeksforgeeks.org/python-classes-and-objects/). It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class that contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

# Python program showing

# abstract base class work

**from** abc **import** ABC, abstractmethod

**class** Animal(ABC):

**def** move(self):

**pass**

**class** Human(Animal):

**def** move(self):

**print**("I can walk and run")

**class** Snake(Animal):

**def** move(self):

**print**("I can crawl")

**class** Dog(Animal):

**def** move(self):

**print**("I can bark")

**class** Lion(Animal):

**def** move(self):

**print**("I can roar")

# Driver code

R **=** Human()

R.move()

K **=** Snake()

K.move()

R **=** Dog()

R.move()

K **=** Lion()

K.move()

**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 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()