Object-Oriented Programming

Class, Objects, Attributes

BY: HOSSEIN HAGHBIN

Object-Oriented language

- Python is an object-oriented language.
- Object-oriented languages help a programmer to reduce the complexity of programs by reusing existing modules or functions.

Class and objects

- The concept of object-oriented programming language is based on class.
- We know that class is another name for type in Python.

It means a programmer can create objects of their own class.

Some inbuilt classes in Python:

- int
- str
- bool
- float
- list
- dict

Python defines how these classes look and behave.

DEFINING CLASSES

Class is another name for type in Python.

- A class may contain data in the form of fields. Fields are also called attributes.
- Codes in the form of functions known as methods.

```
Class Class_Name:
    Initializer
    attributes
    methods()
    Statement(s)
```

PROGRAM 1: Write a simple class program.

```
class Demo:
    pass

D1=Demo()   #Instance or Object of the class Demo
print(D1)

Output:
<__main__.Demo object at 0x029B3150>
```

The output of the print statement is <__main__.Demo object at 0x029B3150>. It tells us the address of the computer's memory where the object D1 is stored.

PROGRAM 2: Write a program to create a simple class and print the message, "Welcome to Object-oriented Programming" and print the address of the instance of the class.

```
class MyFirstProgram:
    print('Welcome to Object-oriented Programming')

C=MyFirstProgram()        #Instance of class.
print(C)

Output:
Welcome to Object-oriented Programming
<__main__.MyFirstProgram object at 0x028B6C90>
```

Adding Attributes to a Class

Let us consider a simple class called Rectangle which defi nes two instance variables length and breadth.

```
Class Rectangle:
      length=0;       #Attribute length
      breadth=0;      #Attribute breadth
```

To create a Rectangle object we will use the following statement.

```
R1 = Rectangle () # Instance of Class
```

Accessing Attributes of an Object

The syntax used to access the attributes of a class is:

```
<object>.<attribute>
```

PROGRAM 3: Write a program to access the attributes of a class.

```
class Rectangle:
    length=0;           #Attribute length
    breadth=0;          #Attribute breadth

R1 = Rectangle()      #Instance of a class
print(R1.length)      #Access attribute length
print(R1.breadth)     #Access attribute breadth

Output
0
0
```

Assigning Value to an Attribute

The syntax used to assign a value to an attribute of an object is

<object>.<attribute> = <Value>

The value can be anything like:

- ✓ a Python primitive,
- ✓ an inbuilt data type,
- ✓ another object etc.

It can even be a function or another class.

PROGRAM 4: Write a program to calculate the area of a rectangle by assigning the value to the attributes of a rectangle, i.e. length and breadth.

```
class Rectangle:
    length=0;
    breadth=0;

R1 = Rectangle ()
print('Initial values of Attribute')
print('Length = ',R1.length)
print('Breadth = ',R1.breadth)
print('Area of Rectangle = ',R1.length * R1.breadth )
R1.length = 20
R1.breadth = 30
print('After reassigning the value of attributes')
print('Length = ',R1.length )
print('Breadth = ',R1.breadth )
print('Area of Rectangle is ',R1.length * R1.breadth)
```

PROGRAM 4: Write a program to calculate the area of a rectangle by assigning the value to the attributes of a rectangle, i.e. length and breadth.

Output

```
Initial values of Attribute
Length   =  0
Breadth  =  0
Area of Rectangle =  0
After reassigning the value of attributes
Length =  20
Breadth =  30
Area of Rectangle is  600
```

PROGRAM 5:

```
class pride():
    founded = '1374'
    country = 'Iran'
    color = 'White'

x=pride()
print(x.color)
print(x.country)
```

Output

```
White
Iran
```

Adding Methods to a Class

The syntax to add methods in a class is:

```
class Class_Name:
    instance variable;      #instance variable with
initialization
    def mthod_name(Self,parameter_list):    #Paramter List is
Optional
        block_of_statements
```

The Self-parameter

To add methods to an existing class, the fi rst parameter for each method should be self. The self-parameter is used in the implementation of the method, but it is not used when the method is called.

Therefore, the self-parameter references the object itself.

PROGRAM 6: Write a program to create a method Display_Message() in a class having the name MethodDemo and display the message, "Welcome to Python Programming"

```
class MethodDemo:
    def Display_Message(self):
        print('Welcome to Python Programming')

ob1 = MethodDemo()  #Instance of a class
ob1.Display_Message() #Calling Method


Output

Welcome to Python Programming
```

The first parameter for each method inside a class should be defined by the name 'self'

PROGRAM 7: Write a program to create a class named Circle. Pass the parameter radius to the method named Calc _ Area() and calculate the area of the circle.

```
import math
class Circle:
    def Calc_Area(self,radius):
        print('radius = ',radius)
        return math.pi*radius**2

ob1 = Circle()
print('Area of circle is ',ob1.Calc_Area(5))
```

Output

```
radius = 5

Area of circle is 78.53981633974483
```

PROGRAM 8: Write a program to calculate the area of a rectangle. Pass the length and breadth of the rectangle to the method named Calc_Rect_Area().

```
class Rectangle:
    def Calc_Area_Rect(self,length,breadth):
    print('length = ',length)
    print('breadth = ',breadth)
    return length*breadth

ob1 = Rectangle()
print('Area of Rectangle is ',ob1.Calc_Area_Rect(5,4))
```

Output

```
length = 5

breadth = 4

Area of rectangle is 20
```

The Self-parameter with Instance Variable

The self can also be used to refer any attribute/member variable or instance variable of the current object from within the instance method.

PROGRAM 9: Here x displays the value of the local variable and self.x displays the value of the instance variable.

```
class Prac:
    x=5
    def disp(self, x):
        x=30
        print('The value of local variable x is ',x)
        print('The value of instance variable x is ',self.x)

ob=Prac()
ob.disp(50)
```

Output

```
The value of local variable x is 30

The value of instance variable x is 5
```

The Self-parameter with Method

The self is also used within methods to call another method from the same class.

PROGRAM 10: Write a program to create two methods, i.e. **Method_A()** and **Method_B()**. Call **Method_A()** from **Method_B()** using self.

```
class Self_Demo:
    def Method_A(self):
        print('In Method A')
        print('wow got a called from A!!!')
    def Method_B(self):
        print('In Method B calling Method A')
        self.Method_A() #Calling Method_A

Q=Self_Demo()
Q.Method_B() #calling Method_B
```

Output

```
In Method B calling Method A
In Method A
wow got a called from A!!!
```

DISPLAY CLASS ATTRIBUTES AND METHODS

There are two ways to determine the attributes in a class. One way is by using the inbuilt function **dir().** The syntax used to display **dir()** attributes is:

```
dir(name_of_class)
or
dir(Instance_of_class)
```

PROGRAM 11: Write a program to display the attributes present in a given class

```
class DisplayDemo:
    Name = ''; #Attribute
    Age = ' '; #Attribute
    def read(self):
        Name=input('Enter Name of student: ')
        print('Name = ',Name)
        Age=input('Enter Age of the Student:')
        print('Age = ',Age)
D1 = DisplayDemo()
D1.read()
>>>(dir( DisplayDemo )
['Age', 'Name', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__','__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__','__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'read']
```

DISPLAY CLASS ATTRIBUTES AND METHODS

An alternate way to display the attributes of a class is by using a special class attribute __dict__. The syntax to display the attributes and methods of an existing class using __dict__ is

```
Class_Name.__dict__
```

PROGRAM 12: Write a program executing __dict__ method on Program 6.

```
class DisplayDemo:
    Name = '';
    Age = ' ';
    def read(self):
        Name=input('Enter Name of student: ')
        print('Name = ',Name)
        Age=input('Enter Age of the Student:')
        print('Age = ',Age)
D1 = DisplayDemo()
D1.read()

>>> DisplayDemo.__dict__
mappingproxy({'read': <function DisplayDemo.read at 0x02E7C978>,
'__weakref__': <attribute '__weakref__' of 'DisplayDemo' objects>,
'__doc__':None, '__dict__': <attribute '__dict__' of 'DisplayDemo'
objects>, '__module__': '__main__', 'Name': '', 'Age': ' '})
```

THE __init__ METHOD (CONSTRUCTOR)

The __init__ method is known as an initializer. It is a special method that is used to initialise the instance variable of an object. The syntax of adding __init__ method to a class is given as follows:

```
class Class_Name:
    def __init__(self): #__init__ method
        ……….
        ……….
```

PROGRAM 13: Write a simple program using the init method

```
class Circle:
    def __init__(self,pi):
        self.pi = pi
    def calc_area(self,radius):
        return self.pi*radius**2

C1=Circle(3.14)
print(' The area of Circle is ',C1.calc_area(5))
```

Output

```
The area of Circle is 78.5
```

Attributes and __init__ Method

Programmers can initialise the value of a member variable or attribute by making use of the __init__ method.

PROGRAM 14: Write a program to initialise the value of the attributes by making use of the init method

```
class Circle:
    pi = 0; #Attribute pi
    radius = 0 #Attribute radius
    def __init__(self):
        self.pi = 3.14
        self.radius = 5
    def calc_area(self):
        print('Radius = ',self.radius)
        return self.pi*self.radius**2

C1=Circle()
print(' The area of Circle is ',C1.calc_area())
```

PROGRAM 15: Write a program to calculate the volume of a box.

```
class Box:
    width = 0; #Member Variables
    height = 0;
    depth = 0;
    volume = 0;
    def __init__(self):
        self.width = 5
        self.height = 5
        self.depth = 5
    def calc_vol(self):
        print('Width = ',self.width)
        print('Height = ',self.height)
        print('depth = ',self.depth)
        return self.width * self.height * self.depth
```

PROGRAM 15: Write a program to calculate the volume of a box.

```
B1=Box()
print(' The Volume of Cube is ',B1.calc_vol())
```

Output

Width = 5

Height = 5

Depth = 5

The Volume of Cube is 125

ACCESSIBILITY

In Python, there are no keywords like **public**, **protected** or **private**. All attributes and methods are **public** by default. There is one way to define private in Python. The syntax to define private attribute and methods is

```
__Attribute
__Methods_Name()
```

To make an attribute and a method private, we need to add two underscores, i.e. "__" in front of the attribute and the method's name. It helps in hiding these when accessed out of class.

PROGRAM 16: Write a program to illustrate the use of private

```
class Person:
    def __init__(self):
        self.Name = 'Bill Gates' #Public attribute
        self.__BankAccNo =10101 #Private attribute
    def Display(self):
        print(' Name = ',self.Name)
        print('Bank Account Number = ',self.__BankAccNo)

P = Person()
#Access public attribute outside class
print(' Name0 = ',P.Name)
P.Display()
#Try to access private variable outside class but fails
print(' Salary = ',P.__BankAccNo)
```

PROGRAM 16: Write a program to illustrate the use of private

```
Output
Name0 = Bill Gates
Name = Bill Gates
Bank Account Number = 10101
Traceback (most recent call last): #Error
    File “C:/Python34/PrivateDemo.py”, line 13, in <module>
        print(' Salary = ',P.__BankAccNo)
AttributeError: 'Person' object has no attribute '__BankAccNo'
```

PASSING AN OBJECT AS PARAMETER TO A METHOD

So far, we have learnt about passing any kind of parameter of any type to methods.

We can also pass objects as parameter to a method.

PROGRAM 17: Write a program to pass an object as parameter to a method.

```
class Test:
    a = 0
    b = 0
    def __init__(self, x , y):
        self.a = x
        self.b = y
    def equals(self, obj):
        if(obj.a == self.a and obj.b == self.b):
            return True
        else:
            return False
```

PROGRAM 17: Write a program to pass an object as parameter to a method.

```
Obj1 = Test(10,20)
Obj2 = Test(10,20)
Obj3 = Test(12,90)
print(' Obj1 == Obj2 ',Obj1.equals(Obj2))
print(' Obj1 == Obj3 ',Obj1.equals(Obj3))

Output
Obj1 == Obj2 True
Obj1 == Obj3 False
```

PROGRAM 18: Using __eq__ method in last programg.

```
class Test:
    a = 0
    b = 0
    def __init__(self, x , y):
        self.a = x
        self.b = y
    def __eq__(self, obj):
        if(obj.a == self.a and obj.b == self.b):
            return True
        else:
            return False
```

PROGRAM 18: Using __eq__ method in last programg.

```
Obj1 = Test(10,20)
Obj2 = Test(10,20)
Obj3 = Test(12,90)
print(' Obj1 == Obj2 ',Obj1==Obj2)
print(' Obj1 == Obj3 ',Obj1==Obj3)

Output
Obj1 == Obj2 True
Obj1 == Obj3 False
```

PROGRAM 19: Write a program to calculate the area of a rectangle by passing an object as parameter to method.

```
class Rectangle:
    def __init__(self, l , w):
        self.length = l
        self.breadth = w
    def __repr__(self):
        return f' Rectangle with length {self.length} and breadth
{self.breadth}'
    def Calc_Area(self, obj):
        print(' Length = ',obj.length)
        print(' Breadth = ',obj.breadth)
        return obj.length * obj.breadth
```

PROGRAM 19: Write a program to calculate the area of a rectangle by passing an object as parameter to method.

```
Obj1 = Rectangle(10,20)
print('The area of Rectangle is ', Obj1.Calc_Area(Obj1))

Output
Length = 10
Breadth = 20
The area of Rectangle is 200
```

Special Methods for Arithmetic Operations

A programmer can overload any arithmetic operation by implementing the corresponding special method.

Operation	*Special Method*	*Description*
X + Y	__add__(self, Other)	Add X and Y
X - Y	__sub__(Self, Other)	Subtract Y from X
X * Y	__mul__(self, Other)	Product of X and Y
X / Y	__truediv__(self, Other)	Y divides X and it shows the quotient as its output
X // Y	__floordiv__(self,Other)	Floored quotient of X and Y
X % Y	__mod__(self, Other)	X mod Y gives a remainder when dividing X by Y
-X	__neg__(self)	Arithmetic negation of X

PROGRAM 20: Write a program to overload the + Operator and perform the addition of two objects.

```
class OprOverloadingDemo:
    def __init__(self,X):
    self.X = X
    def __add__(self,other):
        print(' The value of Ob1 =',self.X)
        print(' The value of Ob2 =',other.X)
        print(' The Addition of two objects is:',end='')
        return ((self.X+other.X))

Ob1 = OprOverloadingDemo(20)
Ob2 = OprOverloadingDemo(30)
Ob3 = Ob1 + Ob2
print(Ob3)
```

Output

```
The value of Ob1 = 20
The value of Ob2 = 30
The Addition of two objects is: 50
```

Note: Ob1 + Ob2 is equivalent to **Ob1.__add__(Ob2)**

Special Methods for Comparing Types

Comparison is not strictly done on numbers. It can be made on various types, such as list, string and even on dictionaries.

Operation	*Special Method*	*Description*
X == Y	__eq__(self, other)	is X equal to Y?
X < Y	__lt__(self, other	is X less than Y?
X <= Y	__le__(self, other)	is X less than or equal to Y?
X > Y	__gt__(self, other)	is X greater than Y?
X >= Y	__ge__(self, other)	is greater than or equal to Y?

PROGRAM 21: Write a program to use special methods and compare two objects.

```
class CmpOprDemo:
    def __init__(self,X):
        self.X = X
    def __lt__(self,other):
        print(' The value of Ob1 =',self.X)
        print(' The value of Ob2 =',other.X)
        print(' Ob1 < Ob2 :',end='')
        return self.X <other.X
    def __gt__(self,other):
        print(' Ob1 > Ob2 :',end='')
        return self.X > other.X
    def __le__(self,other):
        print(' Ob1 <= Ob2 :',end='')
        return self.X <= other.X
```

PROGRAM 21: Write a program to use special methods and compare two objects.

```
Ob1 = CmpOprDemo(20)
Ob2 = CmpOprDemo(30)
print( Ob1 < Ob2 )
print( Ob1 > Ob2 )
print( Ob1 <= Ob2 )
```

Output

```
The value of Ob1 = 20

The value of Ob2 = 30

Ob1 < Ob2 :True

Ob1 > Ob2 :False

Ob1 <= Ob2 :True
```

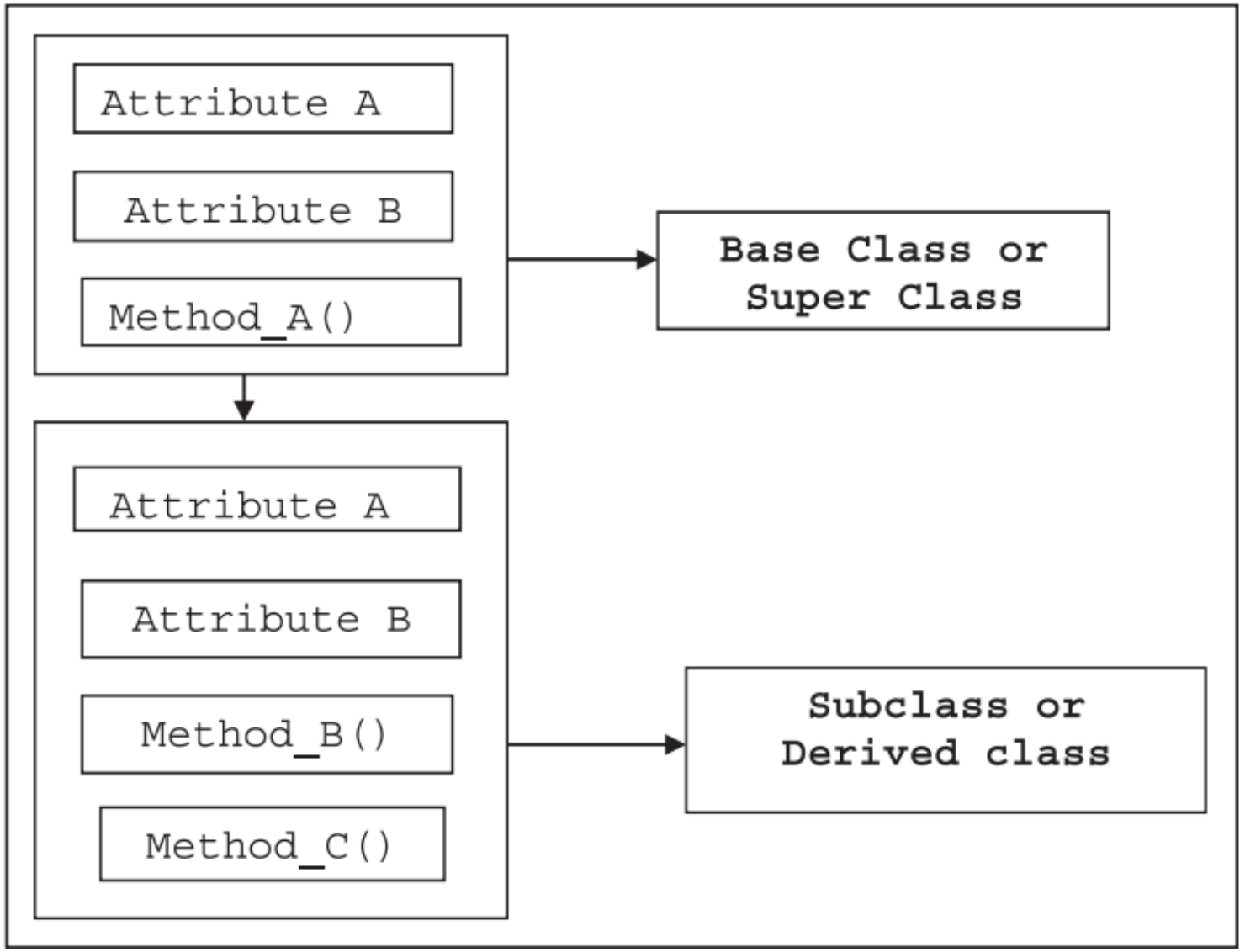
Special Methods for Overloading Inbuilt Functions

Like operators, we can also overload inbuilt functions. Several inbuilt functions can be overloaded in a manner similar to overloading normal operators in Python.

Operation	*Special Method*	*Description*
abs(x)	__abs__(Self)	Absolute value of x
float(x)	__float__(self)	Float equivalent of x
str(x)	__str__(self)	String representation of x
iter(x)	__itr__(self)	Iterator of x
hash(x)	__hash__(self)	Generates an integer hash code for x
len(x)	__len__(self)	Length of x

INHERITANCE

Inheritance is one of the most useful and essential characteristics of object-oriented programming. The existing classes are the main components of inheritance. New classes are created from the existing ones. The properties of the existing classes are simply extended to the new classes. A new class created using an existing one is called a derived class or subclass and the existing class is called a base class or super class.



TYPES OF INHERITANCE

Inheritance can be classified as

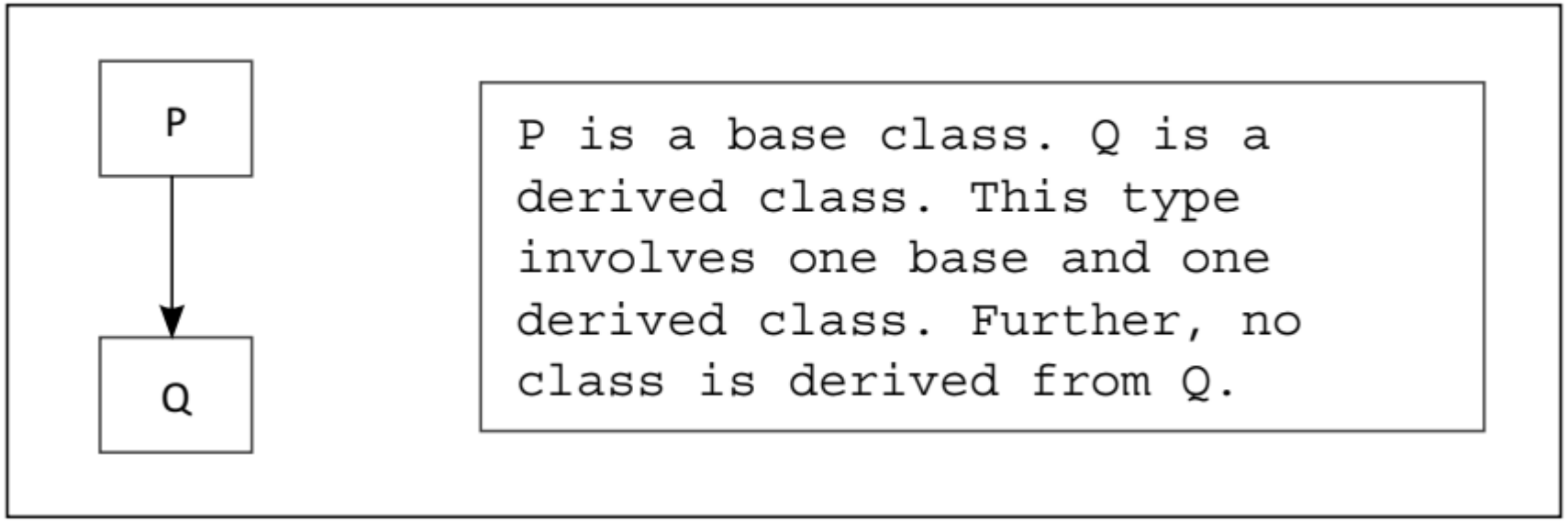
(i) Single inheritance

(ii) Multilevel inheritance

(iii) Multiple inheritance

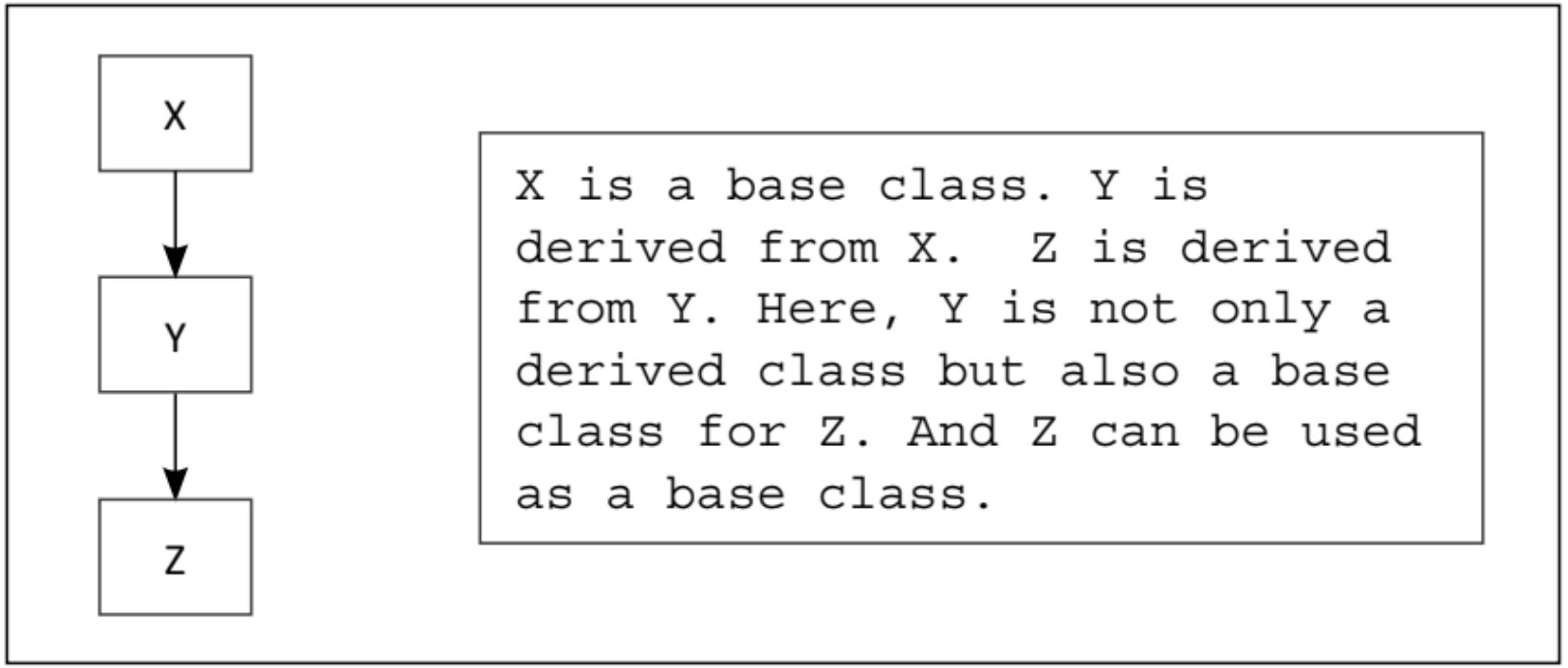
Single inheritance

Only one base class is used for deriving a new class. The derived class is not used as the base class.



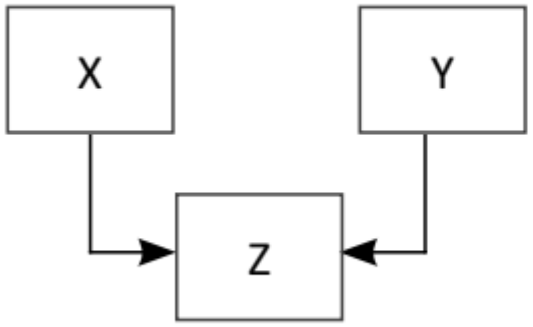
Multilevel inheritance

When a class is derived from another derived class, the derived class acts as the base class.



Multiple inheritance

When two or more base classes are used for deriving a new class, it is called multiple Inheritance.

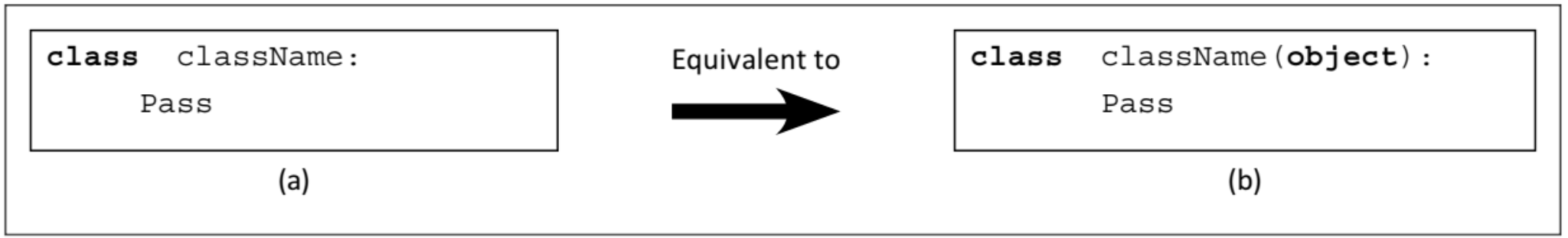



```
X and Y are base classes. Z is
a derived class. Class Z
inherits properties of both X
and Y. Further, Z is not used
as a base class.
```

THE OBJECT CLASS

Every class in Python is derived from the object class. The object class is defined in the Python library. If no inheritance is specified when a class is defined then by default the class is derived from its super class object.

Example



INHERITANCE IN DETAIL

The syntax to inherit single base class in Python is:

```
Class Derived_Class_Name(Single_Base_Class_Name):
        Body_of_Derived_Class
```

The syntax to inherit multiple base classes is:

```
Class Derived_Class_Name(Comma_Seperated_Base_Class_Names):
        Body_of_Derived_Class
```

PROGRAM 22: Write a simple program on inheritance.

```
class A:
    print('Hello I am in Base Class')
class B(A):
    print('Wow!! Great ! I am Derived class')


ob2 = B() #Instance of class B
```

Output:

```
Hello I am in Base Class
Wow!! Great! I am Derived class
```

PROGRAM 23: Write program to create a base class with Point. Defi ne the method **Set_Cordinate(X,Y)**. Defi ne the new class **New_Point**, which inherits the Point class. Also add **draw()** method inside the subclass..

```
Class Point: #Base Class
    def Set_Cordinates(self,X, Y):
        self.X = X
        self.Y = Y
class New_Point(Point): #Derived Class
    def draw(self):
        print(' Locate Point X = ',self.X,' On X axis')
        print(' Locate Point Y = ',self.Y,' On Y axis')
```

PROGRAM 23: Write program to create a base class with Point. Defi ne the method **Set_Cordinate(X,Y)**. Defi ne the new class **New_Point**, which inherits the Point class. Also add **draw()** method inside the subclass..

```
P = New_Point() #Instance of Derived Class
P.Set_Cordinates(10,20)
P.draw()
Output:
Locate Point X = 10 On X axis
Locate Point Y = 20 On Y axis
```

PROGRAM 24: Write a program to inherit attributes of the parent class to a child class

```
class A: # Base Class
    i = 0
    j = 0
    def Showij(self):
        print('i = ',self.i,' j = ',self.j)

class B(A): #Class B inherits attributes and methods of class A
    k = 0
    def Showijk(self):
        print(' i = ',self.i,' j = ',self.j,' k = ',self.k)
    def sum(self):
        print(' i + j + k = ', self.i + self.j + self.k)
```

PROGRAM 24: Write a program to inherit attributes of the parent class to a child class

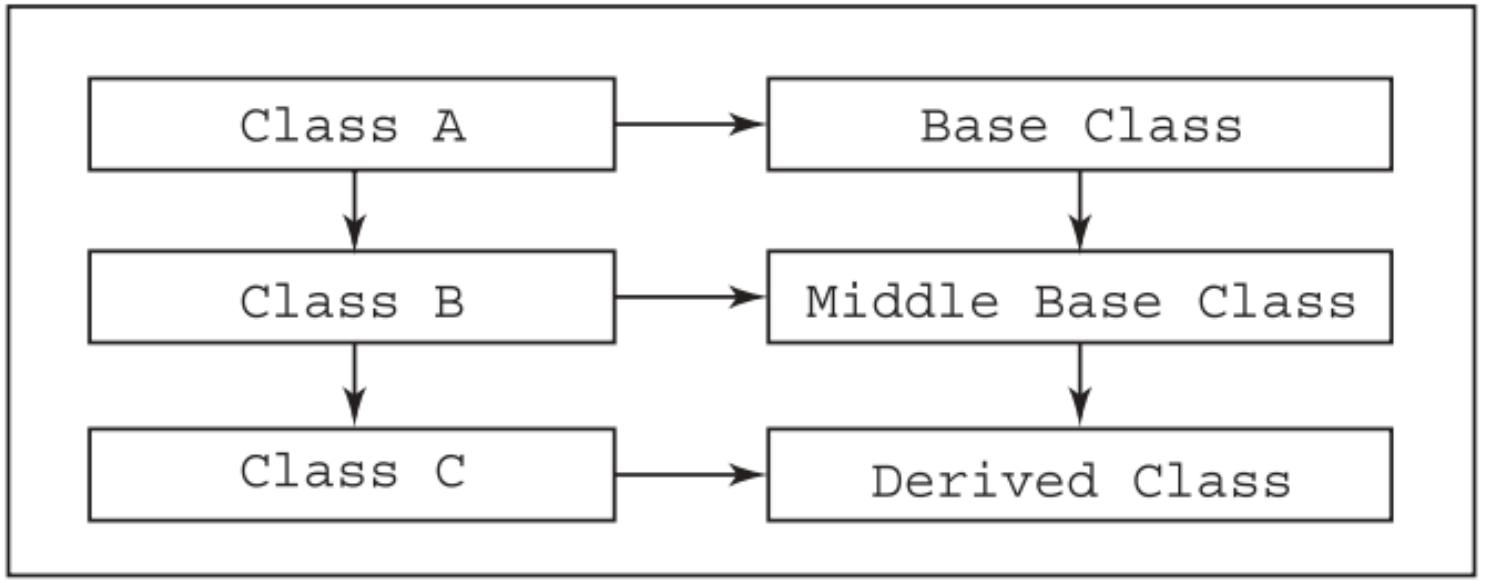
```
Ob1 = A() #Instance of Base class
Ob2 = B() #Instance of Child class
Ob1.i = 100
Ob1.j = 200
print(' Contents of Obj1 ')
Ob1.Showij()
Ob2.i = 100
Ob2.j = 200
Ob2.k = 300
print(' Contents of Obj2 ')
Ob2.Showij() #Sub class Calling method of Base Class
Ob2.Showijk()
print(' Sum of i, j and k in Ob2')
Ob2.sum()
```

PROGRAM 24: Write a program to inherit attributes of the parent class to a child class

```
Output:
Contents of Obj1
i = 100 j = 200
Contents of Obj2
i = 100 j = 200
i = 100 j = 200 k = 300
Sum of i, j and k in Ob2
i + j + k = 600
```

MULTILEVEL INHERITANCE IN DETAIL

The procedure of deriving a class from a derived class is called multilevel inheritance.



PROGRAM 25: Write a simple program to demonstrate the concept of multilevel inheritance.

```
class A: #Base Class
    name = ' '
    age = 0
class B(A): #Derived Class inheriting Base Class A
    height = ' '
class C(B): #Derived Class inheriting his Base Class B
    weight = ' '
    def Read(self):
        print('Please Enter the Following Values')
        self.name=input('Enter Name:')
        self.age = (int(input('Enter Age:')))
        self.height = (input('Enter Height:'))
        self.weight = (int(input('Enter Weight:')))
    def Display(self):
```

PROGRAM 25: Write a simple program to demonstrate the concept of multilevel inheritance.

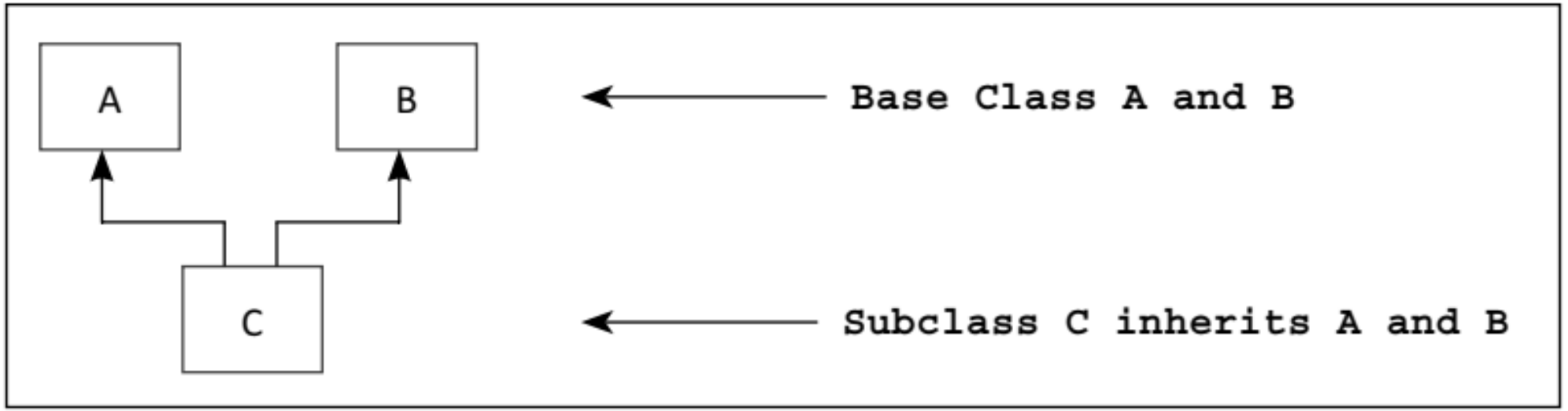
```
        print('Entered Values are as follows')
        print(' Name = ',self.name)
        print(' Age = ',self.age)
        print(' Height = ',self.height)
        print(' Weight = ',self.weight)

B1 = C() #Instance of Class C
B1.Read() #Invoke Method Read
B1.Display() #Invoke Method Display
```

PROGRAM 25: Write a simple program to demonstrate the concept of multilevel inheritance.

```
Output
Please Enter the Following Values
Enter Name: Amit
Enter Age:25
Enter Height:5,7'
Enter Weight:60
Entered Values are as follows
Name = Amit
Age = 25
Height = 5,7'
Weight = 60
```

MULTIPLE INHERITANCE IN DETAIL

When two or more base classes are used for derivation of a new class, it is called multiple inheritance.



PROGRAM 26: Write a simple program to demonstrate multiple inheritance

```
class A: #Base Class A
    a = 0
class B: #Other Base Class B
    b = 0
class C(A,B): #Inherit A and B to create New Class C
    c = 0
    def Read(self):
        self.a =(int(input('Enter the Value of a:')))
        self.b =(int(input('Enter the value of b:')))
        self.c =(int(input('Enter the value of c:')))
    def display(self):
        print(' a = ',self.a)
        print(' b = ',self.b)
        print(' c = ',self.c)
```

PROGRAM 26: Write a simple program to demonstrate multiple inheritance

```
Ob1 = C() #Instance of Child Class
Ob1.Read()
Ob1.display()

Output

Enter the Value of a:10
Enter the value of b:20
Enter the value of c:30
a = 10
b = 20
c = 30
```

PROGRAM 27: Write a program to calculate the volume of Box using the **init()** method

```
Class Box:
    width = 0
    height = 0
    depth = 0
    def __init__(self,W,H,D):
        self.width = W
        self.height = H
        self.depth = D
    def volume(self):
        return self.width * self.height * self.depth
```

PROGRAM 27: Write a program to calculate the volume of Box using the **init()** method

```
class ChildBox(Box):
    weight = 0
    def __init__(self,W,H,D,WT):
        self.width = W
        self.height = H
        self.depth = D
        self.weight = WT
    def volume(self):
        return self.width * self.height * self.depth
```

PROGRAM 27: Write a program to calculate the volume of Box using the **init()** method

```
B1 = ChildBox(10,20,30,150)
B2 = ChildBox(5,4,2,100)
vol = B1.volume()
print(' ----- Characteristics of Box1 ---- ')
print(' Width = ',B1.width)
print(' height = ',B1.height)
print(' depth = ',B1.depth)
print(' Weight = ',B1.weight )
print(' Volume of Box1 = ',vol)
print(' ----- Characteristics of Box2---- ')
print(' Width = ',B2.width)
print(' height = ',B2.height)
print(' depth = ',B2.depth)
print(' Weight = ',B2.weight )
vol = B2.volume()
print(' Volume of Box2 =',vol)
```

PROGRAM 27: Write a program to calculate the volume of Box using the **init()** method

```
Output
----- Characteristics of Box1 ----
Width = 10
height = 20
depth = 30
Weight = 150
Volume of Box1 = 6000
----- Characteristics of Box2----
Width = 5
height = 4
depth = 2
Weight = 100
Volume of Box2 = 40
```

USING super()

Consider the following program:

```
class Demo:
    a = 0
    b = 0
    c = 0
    def __init__(self,A,B,C):
        self.a = A
        self.b = B
        self.c = C
    def display(self):
        print(self.a,self.b,self.c)

class NewDemo(Demo):
    d = 0
    def __init__(self,A,B,C,D):
        self.a = A
        self.b = B
        self.c = C
        self.d = D
    def display(self):
        print(self.a,self.b,self.c,self.d)
```

USING super()

Consider the following program:

```
B1 = Demo(100,200,300)
print(' Contents of Base Class')
B1.display ()
D1=NewDemo(10,20,30,40)
print(' Contents of Derived Class')
D1.display()
```

Output

```
Contents of Base Class
100 200 300
Contents of Derived Class
10 20 30 40
```

USING super()

In the above program, the classes derived from the base class **Demo** were not implemented efficiently or robustly. For example, the derived class **NewDemo** explicitly initialises the value of **A, B** and **C**, fields of the Base class. The same duplication of code is found while initializing the same fields in the base class **Demo**, which is inefficient. This implies that a subclass must be granted access to the members of a super class.

USING super()

Therefore, whenever a subclass needs to refer to its immediate **super** class, a programmer can do so by using **super**. The super is used to call the constructor, i.e. the **__init__** method of the super class.

Super to Call Super Class Constructor

Any subclass can call the constructor, i.e. the **__init__** method defned by its super class by making use of super.

```
super().__init__(Parameters_of_Super_class_Constructor)
```

PROGRAM 29: Use super() and call the constructor of the base class

```
class Demo:
	def __init__(self,A,B,C):
		self.a = A
		self.b = B
		self.c = C
	def display(self):
		print(self.a,self.b,self.c)

class NewDemo(Demo):
	def __init__(self,A,B,C,D):
		self.d = D
		super().__init__(A,B,C) #Super to call Super class
	def display(self):
		print(self.a,self.b,self.c,self.d)
```

PROGRAM 29: Use super() and call the constructor of the base class

```
B1 = Demo(100,200,300)
print(' Contents of Base Class')
B1.display ()
D1=NewDemo(10,20,30,40)
print(' Contents of Derieved Class')
D1.display()
```

Output

```
Contents of Base Class
100 200 300
Contents of Derived Class
10 20 30 40
```

METHOD OVERRIDING

In class hierarchy, when a method in a sub class has the same name and same header as that of a super class then the method in the sub class is said to override the method in the super class. When an overridden method is called, it always invokes the method defined by its subclass. The same method defined by the super class is hidden.

PROGRAM 30: Write a program to show method overriding.

```
class A: #Base Class
      i = 0
      def display(self):
            print(' I am in Super Class')

class B(A): #Derived Class
      i = 0
      def display(self): #Overridden Method
            print(' I am in Sub Class')

D1 = B()
D1.display()
```

Programmer can make use of super to access the overridden methods. The syntax to call the overridden method that is defined in super class is

```
super().method_name
```

PROGRAM 31: Write a program to show method overriding.

```
class A: #Base Class
        i = 0
        def display(self):
                print(' I am in Super Class')
class B(A): #Super Class
        i = 0
        def display(self): #Overriden Method
                print(' I am in Sub Class')
                super().display() #Call Display method of Base class

D1 = B() #Instance of sub class
D1.display()
```

Output:

I am in Sub Class
I am in Super Class

PROGRAM 32: Program to override Display() method in multiple inheritance.

```
class A(object):
    def Display(self):
    print(" I am in A")

class B(A):
    def Display(self):
        print(" I am in B")
        A.Display(self) # call the parent class method too

class C(A):
    def Display(self):
        print(" I am in C")
        A.Display(self)
```

PROGRAM 32: Program to override Display() method in multiple inheritance.

```
class D(B, C):
      def Display(self):
            print(" I am in D")
            B.Display(self)
            C.Display(self)
Ob = D()
Ob.Display()
```

Output

```
I am in D
I am in B
I am in A
I am in C
I am in A
```

The problem with the above method is that **A.Display** method has been called twice. If we have a complex tree of multiple inheritance then it is very difficult to solve this problem by hand. We have to keep track of which super classes have already been called and avoid calling them a second time. Therefore, to solve the above problem, we can make use of super. Consider the same program with some modifications

PROGRAM 33: Program to override Display() method in multiple inheritance.

```
class A(object):
    def Display(self):
        print(" I am in A")
class B(A):
    def Display(self):
        print(" I am in B")
        super().Display() # call the parent class method too
class C(A):
    def Display(self):
        print(" I am in C")
        super().Display()
```

PROGRAM 33: Program to override Display() method in multiple inheritance.

```
class D(B, C):
      def Display(self):
            print(" I am in D")
            super().Display()

Ob = D()
Ob.Display()
```

Output

```
I am in D
I am in B
I am in C
I am in A
```