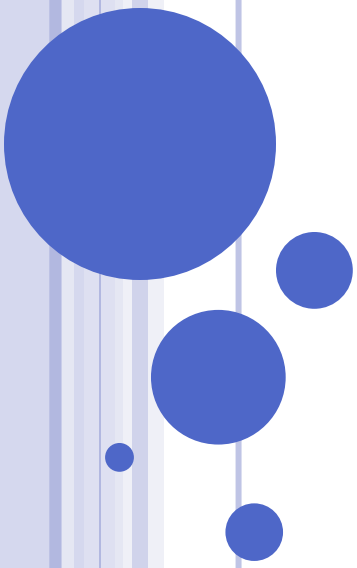


CHAPTER 5

CLASSES AND OBJECT ORIENTED PROGRAMMING

INTRODUCTION

- To write programs in Python using classes to organize programs around modules and data abstractions.
- Context of Object oriented Programming.
- The key to object oriented programming is thinking about objects as collections of both data and the methods that operate on that data.

INTRODUCTION

- Class is a model or plan to create objects.
- We write a class with attributes and actions of objects.
- Attributes are represented by variables and actions are performed by methods.
- Difference between a function and a method.
  - A function written inside a class is called a method.
  - A method is called using one of the following ways:
    - **classname.methodname()**
    - **instancename.methodname()**

CREATING A CLASS

- **General format of class:**
  - **class Classname(object):**
    - **attributes**
    - **def_init_(self):**
    - **def method1():**
    - **def method2():**
- **CREATING A CLASS:**
  - A class is created with the keyword *class* and then writing the *Classname.*
  - After the Classname, '*object*' is written inside the Classname.
  - This 'object' represents the base class name from which all classes in Python are derived.
  - Even our own classes are also derived from 'object' class.

Creating a class

- 'attributes' are nothing but variables that contains data.
- _init_(self) is a special method to initialize the variables.
- method1() and method2() etc.. are methods that are intended to process variables.
- Example take 'Student' class, write code in the class that specifies the attributes and actions performed by any student.
- A student has attributes like name, age, marks etc.
- These attributes should be written inside the Student class as variables.

Creating a class

- A student can perform actions like talking , writing , reading etc.
- These actions should be represented by methods in the Student class.

```
class Student():
    def _init_(self):
        self.name='Krish'
        self.age=20
        self.marks=500

    def talk(self):
        print('Hi, I am ', self.name)
        print('My age is ', self.age)
        print('My marks are ', self.marks)
s1 = Student()
s1.talk()
```

Creating a class

- Keyword class is used to declare a class.
- 'Student' is our class name.
- A class should start with capital letter hence 'S' is capital in 'Student'.
- Since in Python we cannot declare variables, we have written the variables inside a special method _init_().
- This method is used to initialize the variables.
- Hence the 'init'.
- The method has two underscores before and after.
- This indicates that this method is internally defined and we cannot call it explicitly.

Creating a class

- ‘self’ is a variable that refers to the current class instance.
- When we create an instance for the Student class, a separate memory block is allocated on the heap and that memory location is by default stored in ‘self’.
- The instance contains the variables name, age, marks which are called instance variables.
- To refer to instance variables, use dot operator along with self as ‘self.name’, ‘self.age’, ‘self.marks’.

Creating a class

- The method talk() takes the 'self' variable as parameter.
- This method displays the values of the variables by referring them using 'self'.
- The methods that act on instances of a class are called instance methods.
- Instance methods use 'self' as the first parameter that refers to the location of the instance in the memory.
- Since, instance methods know the location of instance they can act on the instance variables.
- The two methods _init_(self) and talk(self) are called instance methods.

Creating a class

- In the Student class, a student is talking to us through talk() method.
- Writing a class like this is not sufficient.
- It should be used.
- To use a class, we should create an instance (object) to the class.
- Instance creation represents allocating memory necessary to store the actual data of the variables.

Creating a class

- To use a class, we should create an instance to the class.
- Instance creation represents allotting memory necessary to store the actual data of the variables.
- Syntax to create instance:
  - **instancename = Classname()**
- Eg: s1 = Student()
- s1 is nothing but the instance name.

Creating a class

- When we create an instance the following steps take place internally:
  - A block of memory is allocated on heap.
  - Memory is decided from the attributes and methods available from the class.
  - After allocating the memory block, the special method by the name __init__(self) is called internally.

Creating a class

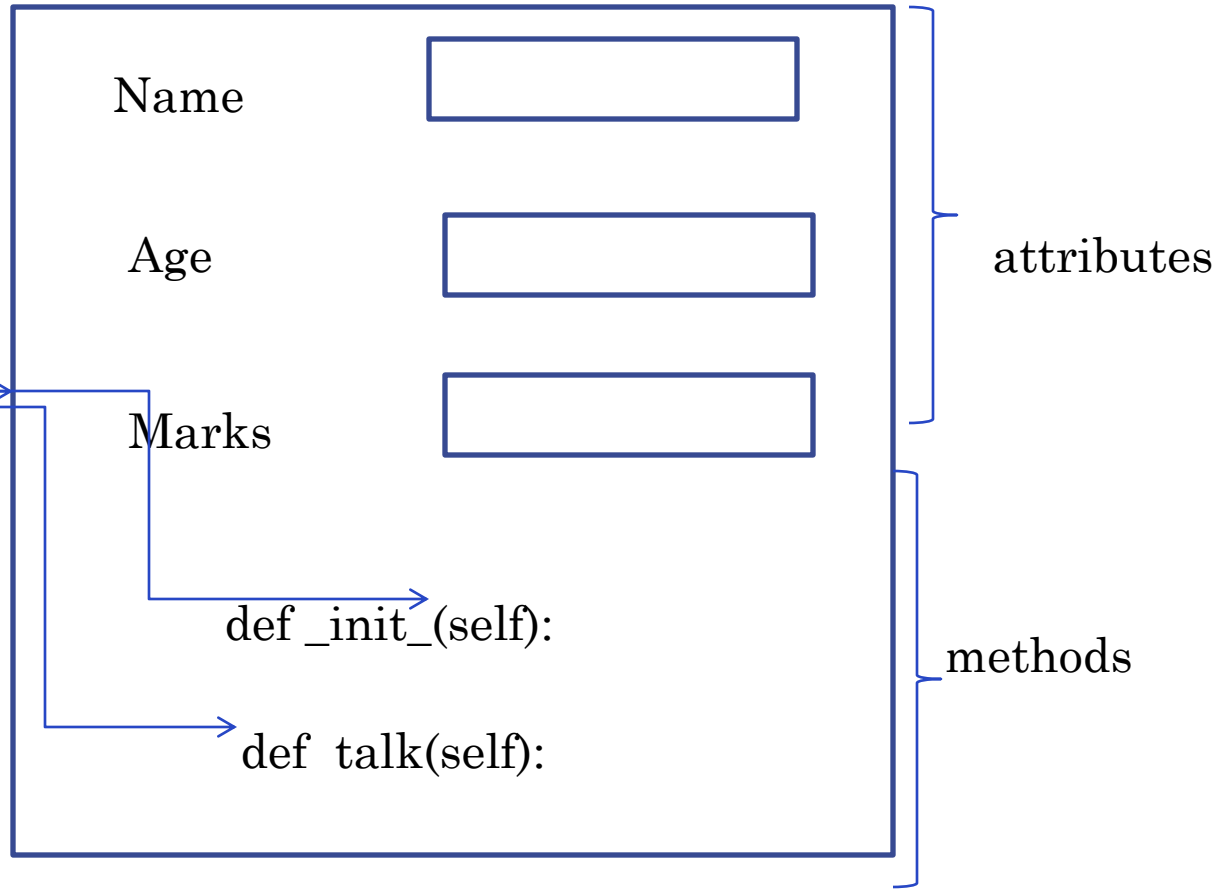
- When we create an instance the following steps take place internally:
  - This method stores the initial data into the variables. Since this method is used to construct the instance, it is called ‘constructor’.
  - Finally, the allocated memory location address of the instance is returned into ‘s1’ variable.
  - To see this memory location in decimal number format, use id() function as id(s1).

Creating a class

- ‘s1’ refers to the instance of the Student class.
- Any variables and methods in the instance can be referenced by ‘s1’ using dot operator.
  - s1.name
  - s1.age
  - s1.marks
  - s1.talk()
- The dot operator takes the instance name at its left and the member of the instance at its right hand side.

Creating a class

- s1()



Creating a class

- Python program to define Student class and create an object to it. Call the method and display the student's details.

```
[6]  class Student:
       def __init__(self):
         self.name = 'Krish'
         self.age = 20
         self.marks = 100

       def talk(self):
         print('Hi i am ',self.name)
         print('My age is',self.age)
         print('My marks are',self.marks)

     s1=Student()
     s1.talk()
```

```
Hi i am  Krish
My age is 20
My marks are 100
```

THE SELF VARIABLE

- 'self' is a default variable that contains the memory address of the instance of the current class.
- 'self' refers to all current instance variables and instance methods.
- When an instance to the class is created, the instance name contains the memory location of the instance.
- This memory location is internally passed to 'self'.

THE SELF VARIABLE

- Eg: s1 = Student()
- ‘s1’ contains the memory address of the instance.
- This memory address is internally passed to ‘self’ variable.
- ‘self’ knows the memory address of the instance, it can refer to all the members of the instance.
- ‘self’ is used in two ways:
  - Used as first parameter in the constructor as:
    - **def __init__(self):**
  - Used as first parameter in the instance method :
    - **def talk(self):**

CONSTRUCTOR

- A constructor is a special method used to initialize the instance variables of a class.
- First parameter of the constructor will be ‘self’ variable that contains the memory address of the instance.
- First parameter of constructor will be ‘self ‘ variable that contains memory address of the instance.
- A constructor is called at the time of creating an instance.

CONSTRUCTOR

- Python program to create Student class with a constructor having more than one parameter.

```
class Student:
  def __init__(self,n = '.',m=0):
    self.name = n
    self.marks = m
  def display(self):
    print('Hi',self.name)
    print('Your marks',self.marks)

s=Student()
s.display()
print('-----------------')
s1=Student('pinky',100)
s1.display()
print('-----------------')
```

```
Hi .
Your marks 0
-----------------
Hi pinky
Your marks 100
-----------------
```

CONSTRUCTOR

- Constructor does not create an instance.
- Only initialize or store the beginning values into the instance variables.
- Constructor is called only once at the time of creating an instance.
- If 3 instances are created for a class, the constructor will be called once per each instance, thus it is called 3 times.

TYPES OF VARIABLES

- Variables written inside a class are of 2 types:
  - **Instance Variables**
  - **Class Variables or Static Variables**
- **Instance Variables:**
  - Variables whose separate copy is created in every instance.
  - Eg: 'x' is an instance variable and if we create 3 instances, there will be 3 copies of 'x' in 3 instances.
  - When we modify the copy of 'x' in any instance, it will not modify the other two copies.
  - Instance variables are defined and initialized using a constructor with 'self' parameter .
  - To access the instance variables, use **self.variable**
  - To access instance variables from outside class as instancename.variable
  - Eg: s1.x

TYPES OF VARIABLES

- Python program to understand instance variables

```
[ ]  class Sample:
       def __init__(self):
         self.x=10

       def modify(self):
         self.x+=1

     s1=Sample()
     s2=Sample()
     print('x in s1= ',s1.x)
     print('x in s2= ',s2.x)

     s1.modify()
     print('x in s1= ',s1.x)
     print('x in s2= ',s2.x)
```

```
x in s1=  10
x in s2=  10
x in s1=  11
x in s2=  10
```

TYPES OF VARIABLES

- Class Variables or Static Variables:
  - Variables whose single copy is available to all the instances of the class.
  - If the copy of instance variable is modified in one instance , it will modify all the copies n other instances.
  - Eg: ‘x ‘ is a class variable and if we create 3 instances.If its value is modified in one instance the same copy of ‘x’ is passed to 3 instances.
  - When we modify the copy of ‘x’ in any instance using a class method, the modified copy is sent to th eother two instances.

TYPES OF VARIABLES

- Class method contains first parameter ‘cls’ with which we can access the class variables.
- Class variables are defined directly in the class.
- Access class variables using class methods as **class.variable.**
- To access calss variables from outside the class, use **classname.variable.**

TYPES OF VARIABLES

- Python program to understand class variables or static variables.

```
[ ]  class Sample:
       x =10

       def modify(cls):
         cls.x+=1

     s1 = Sample()
     s2 = Sample()
     print('x in s1= ',s1.x)
     print('x in s2= ',s2.x)

     s1.modify()
     print('x in s1= ',s1.x)
     print('x in s2= ',s2.x)
```

```
x in s1=  10
x in s2=  10
x in s1=  11
x in s2=  10
```

TYPES OF METHODS

- Purpose of method is to process the variables provided in the class
- Variables declared in the class are called class variables and the variables declared in the constructor are called instance variables.
- Methods are classified into 3 types:
  - **Instance Methods**
    - Accessor Methods
    - Mutator Methods
  - Class Methods
  - Static Methods

TYPES OF METHODS

- Instance Methods:
  - Act upon the instance variables of the class.
  - Bound to instances or objects and hence called as **instancename.method().**
  - Instance methods need to know the memory address of the instance.
  - Provided through 'self' variable by default as first parameter for the instance method.

TYPES OF METHODS

- Python program using Student class with instance methods to process the data of several students.

```
class Student:
  def __init__(self,n='',m=0):
    self.name=n
    self.marks=m
  def display(self):
    print('Hi',self.name)
    print('Your marks',self.marks)
  def calculate(self):
    if(self.marks>=600):
      print('First Grade')
    elif(self.marks>=500):
      print('Second Grade')
    elif(self.marks>=350):
      print('Third grade')
    else:
      print('You are failed')
n=int(input('How many students?'))
i=0
while(i<n):
  name = input('Enter name: ')
  marks = int(input('Enter marks: '))

  s = Student(name,marks)
  s.display()
  s.calculate()
  i+=1
```

TYPES OF METHODS

- Instance methods are of two types:
  - **Accessor Methods**
  - **Mutator Methods**
- **Accessor Methods:**
  - Access or read data of the variables.
  - Do not modify the data in the variables.
  - Written in form of getXX() and they are called getter methods.
- **Mutator Methods:**
  - Not only read the data but also modify them.
  - Written in the form of setXX() and called as setter methods.

TYPES OF METHODS

- Python program to store data into instances using mutator methods and retrieve data from instances using accessor methods.

```
class Student:
  def setname(self,name):
    self.name=name
  def getname(self):
    return self.name
  def setmarks(self,marks):
    self.marks=marks
  def getMarks(self):
    return self.marks
n=int(input('How many students: '))

i=0
while(i<n):
  s=Student()
  name=input('Enter name ')
  s.setname(name)
  marks=int(input('Enter marks: '))
  s.setmarks(marks)
  print('Hi',s.getname())
  print('Your marks', s.getMarks())
  i+=1
```

TYPES OF METHODS

- **Class Methods:**
  - Acts on class level.
  - Class methods are methods which act on the class variables or static variables.
  - Parameter of class method is ‘cls’ which refers to class itself.
  - **‘cls.var’** is the format to refer to the class variable.
  - Class methods are called using **classname.method().**

TYPES OF METHODS

- Python program to use class method to handle the common feature of all the instances of Bird class.

```
class Bird:
  wings = 2
  @classmethod
  def fly(cls,name):
    print('{} flies with {} wings'.format(name, cls.wings))

Bird.fly('Sparrow')
Bird.fly('Pigeon')
```

```
Sparrow flies with 2 wings
Pigeon flies with 2 wings
```

TYPES OF METHODS

- Static Methods:
  - Static methods when the processing is at the class level but we need not involve the class or instances.
  - Example: Setting environmental variables, counting the number of instances of the class or changing an attribute in another class are the tasks handled by static methods.
  - Used to accept values process them and return the result.
  - Static methods are called in the form of classname.method().

TYPES OF METHODS

- Python program to create a static method that counts the number of instances created for a class.

```
[33] class Myclass:
       n=0
       def __init__(self):
         Myclass.n=Myclass.n+1
       def noobjects():
         print('No.of.objects created: ',Myclass.n)
     obj1 = Myclass()
     obj2 = Myclass()
     obj3 = Myclass()
     Myclass.noobjects()
```

```
No.of.objects created:  3
```

TYPES OF METHODS

- Python program to create a static method that calculates the square root value of a given number.

```
import math
class Sample:
  @staticmethod
  def calculate(x):
    result = math.sqrt(x)
    return result
num=float(input('Enter a number: '))
res=Sample.calculate(num)
print('The square root of {} is {:.2f}'.format(num, res))
```

```
Enter a number: 64
The square root of 64.0 is 8.00
```

TYPES OF METHODS

- **Static Methods:**

```
import sys
class Bank(object):
  def __init__(self,name,balance=0.0):
    self.name=name
    self.balance=balance
  def deposit(self,amount):
    self.balance+=amount
    return self.balance
  def withdraw(self,amount):
    if amount>self.balance:
      print('Balance is less so no withdrawal')
    else:
      self.balance-=amount
    return self.balance
name=input('enter name: ')
b = Bank(name)
```

```
while(True):
  print('d-Deposit,w-Withdraw,e-Exit')
  choice = input('Enter your choice: ')
  if choice =='e' or choice =='E':
    sys.exit()
  amt = float(input('Enter amount: '))

  if choice == 'd' or choice =='D':
    print('Balance after deposit: ',b.deposit(amt))
  elif choice == 'w' or choice == 'W':
    print('Balance after withdrawal: ',b.withdraw(amt))
```

INHERITANCE

INHERITANCE

- When a programmer develops a class, he will use its features by creating an instance to it.
- When another programmer wants to create another class which is similar to the class already created, then he need not create the class from the scratch.
- Use the features of the existing class in creating his own class.

INHERITANCE

- Python program to create Teacher class and store it in teacher.py module

```
class Teacher:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self,name):
        self.name=name

    def getname(self):
        return self.name

    def setaddress(self,address):
        self.address = address

    def getaddress(self):
        return self.address

    def setsalary(self, salary):
        self.salary = salary

    def getsalary(self):
        return self.salary
```

INHERITANCE

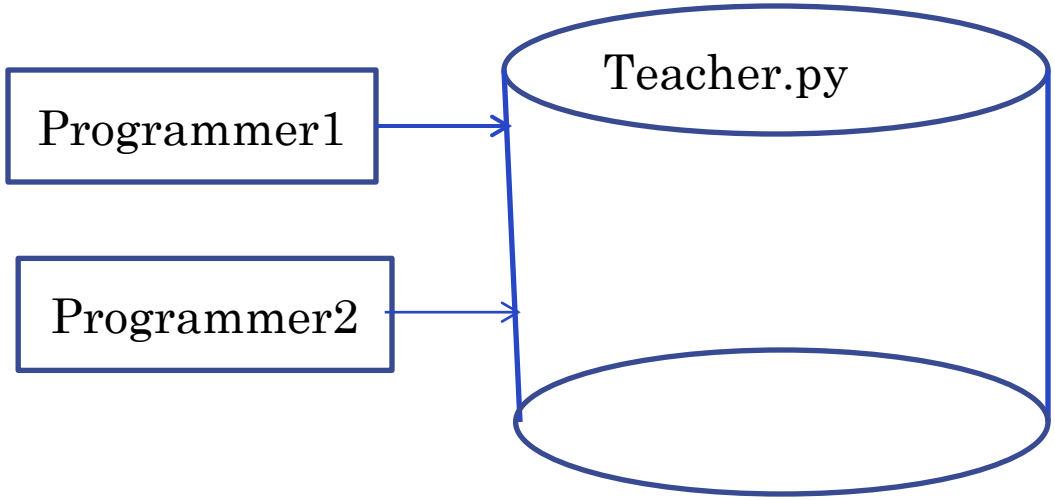
- When the programmer wants to use this Teacher class that is available in teacher.py file he can simply import this class into his program.

```
from teacher import Teacher
t = Teacher()

t.setid(10)
t.setname('Dhanlaxmi')
t.setaddress('SCET')
t.setsalary(250000.00)

print('id= ', t.getid())
print('name= ', t.getname())
print('address= ', t.getaddress())
print('salary= ', t.getsalary())
```

```
In [2]: %run "D:/Python Programs/inh.py"
id=  10
name=  Dhanlaxmi
address=  SCET
salary=  250000.0
```

INHERITANCE

- Once the Teacher class is created, the programmer stored teacher.py program in a central database that is available to all the members of the team.



- The Teacher class is available through the module teacher.py

INHERITANCE

- Another programmer in the same team wants to create a Student class.
- He is planning the Student class without considering the Teacher class.

```
class Student:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self,name):
        self.name=name

    def getname(self):
        return self.name

    def setaddress(self,address):
        self.address = address

    def getaddress(self):
        return self.address

    def setmarks(self,marks):
        self.marks = marks

    def getmarks(self):
        return self.marks
```

INHERITANCE

- Python program to use the Student class which is already available in student.py

```
from student import Student
s = Student()

s.setid(100)
s.setname('Krish')
s.setaddress('JHASV')
s.setmarks(500)

print('id= ', s.getid())
print('name= ', s.getname())
print('address= ', s.getaddress())
print('marks= ', s.getmarks())
```

```
In [5]: %run "D:/Python Programs/inh1.py"
id=  100
name=  Krish
address=  JHASV
marks=  500
```

INHERITANCE

- If we compare both Teacher and Student class, we can understand that 75% of the code is same in both the classes.
- Most of the code being planned by the second programmer in his student class is already available in the Teacher class.
- Instead of creating a new class altogether, we can reuse the code which is already available.

INHERITANCE

- Python program to create Student class by deriving it form the Teacher class.

```
from teacher import Teacher
class Student1(Teacher):
    def setmarks(self,marks):
        self.marks = marks

    def getmarks(self):
        return self.marks
s = Student1()
s.setid(100)
s.setname('Krish')
s.setaddress('JHASV')
s.setmarks(500)

print('id= ',s.getid())
print('name= ',s.getname())
print('address= ',s.getaddress())
print('marks= ',s.getmarks())
```

```
In [4]: %run -i "D:/Python Programs/student1.py"
id=  100
name=  Krish
address=  JHASV
marks=  500
```

INHERITANCE

- In the first statement we are importing Teacher class from teacher module so that the Teacher class is now available to this program.
- Then we are creating Student class as:
  - Class Student1(Teacher)
- This means Student1 class is derived from Teacher class.
- So, all members of Teacher class are available to the Student class.
- We can use them without rewriting them in the Student class.

INHERITANCE

- Two methods that are needed by the Student class but not available in the Teacher class:
  - def setmarks(self,marks):
  - def getmarks(self):
- To use the Student class create an instance to the Student class and call the methods.
- We can say that we have created Student class from Teacher class.
- This is called Inheritance.
- The original class i.e Teacher class is called base class or super class
- Newly created class i.e the Student class is called the sub class or derived class.

INHERITANCE

- **Inheritance:**
  - Deriving new classes from the existing classes such that the new classes inherit all the members of the existing classes inherit all the members of the existing classes is called inheritance.
- Syntax for inheritance:
  - **Class Subclass(Baseclass):**
- When an object to Student class is created, it contains a copy of Teacher class within it.
- There is a relation between Student and Teacher class objects.
- All the members of Teacher class as well as Student class are available in the Student class object.

INHERITANCE

- In inheritance, we always create only the sub class object.
- We do not create super class object.
- Since all the members of both the super and sub class are available to sub class, when we create an object, we can access the members of both the super and sub class.
- But if we create an object to super class, we can access only the super class members and not the sub class members.

Constructors in inheritance

- In the previous program we inherited Student class from Teacher class.
- All the methods and variables in those methods of the Teacher class are accessible to the Student class.
- **Question? Are the constructors of the base class accessible to the sub class or not?**
  - Example a Father class has constructor where a variable 'property; is declared and initialized with some value.
  - When Son is created from Father class, this constructor is by default available to Son class.
  - When we call the method of the super class using sub class object, it will display the value of the 'property' variable.

Constructors in inheritance

- A Python program to access the base class constructor from base class.

```
class Father:
    def __init__(self):
        self.property=800000.00

    def display_property(self):
        print('Father\'s property= ' , self.property)

class Son(Father):
    pass

s = Son()
s.display_property()
```

```
In [22]: %run -i "F:/Python Programs/inh.py"
("Father's property= ", 800000.0)
```

- Conclusion: Like variables and methods, the constructors in the super class are also available to the sub class object.

Overriding superclass constructors and methods

- When the programmer writes a constructor in the sub class, the super class constructor is not available to the sub class.
- **Only the sub class constructor is accessible from the sub class object.**
- **So, the sub class constructor is replacing the super class constructor.**
- **This is called *constructor overriding.***
- If we write a method with exactly same name as that of super class method, it will override super class method.

Overriding superclass constructors and methods

- A Python program to override super class constructor and method in sub class.

```
class Father:
    def __init__(self):
        self.property=800000.00

    def display_property(self):
        print('Father\'s property= ' , self.property)

class Son(Father):
    def __init__(self):
        self.property = 200000.00

    def display_property(self):
        print('Child\'s Property=' , self.property)

s = Son()
s.display_property()
```

```
In [24]: %run -i "F:/Python Programs/inh1.py"
("Child's Property=", 200000.0)
```

Overriding superclass constructors and methods

- So, how to call the super class constructor so that we can access the father's property from the Son class?

- We should **call the constructor of the super class from the constructor of the sub class using the super() method.**

SUPER() METHOD

- super() is a built-in method used to call the super class constructor or methods from the sub class.
- super() is a built-in method in Python that contains the history of super class methods.
- Use super() to refer to super class constructor and methods from a sub class.
- super() can be used as:
  - **super().__init__()** #call super class constructor
  - **super().__init__()(arguments)** #call super class constructor and pass arguments
  - **super.method()** # call super class method

CONSTRUCTORS IN INHERITANCE

- A Python program to call the super class constructor in the sub class using super().

```
class Father(object):
    def __init__(self, property=0):
        self.property=property

    def display_property(self):
        print('Father\'s property= ' , self.property)

class Son(Father):
    def __init__(self, property1=0, property=0):
        super(Son,self).__init__(property)
        self.property1=property1

    def display_property(self):
        print('Child\'s Property=' , self.property1 + self.property)

s = Son(200000.00 , 800000.00)
s.display_property()
```

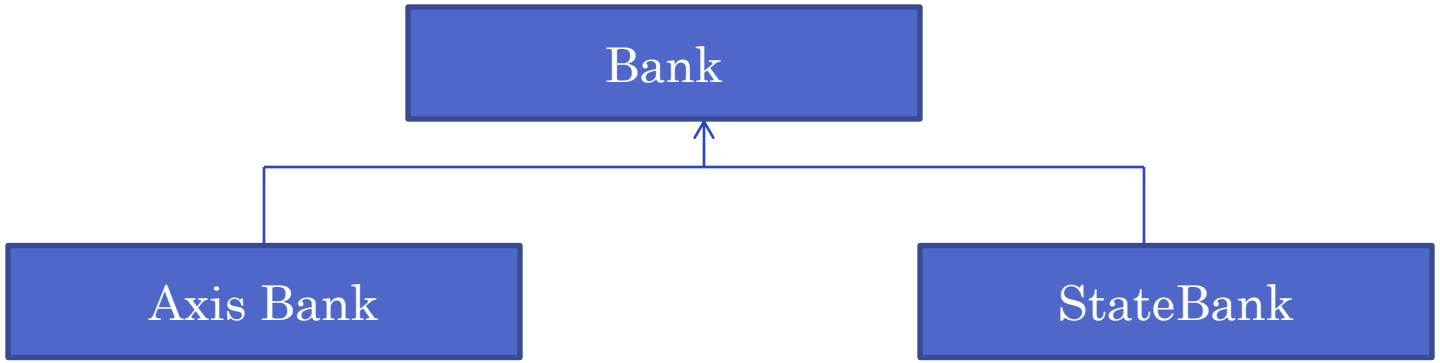
```
In [51]: %run -i "F:/Python Programs/supinherit.py"
("Child's Property=", 1000000.0)
```

TYPES OF INHERITANCE

- There are mainly 2 types of inheritance available:
  - **1. Single Inheritance**
  - **2. Multiple Inheritance**
- **Single Inheritance:**
  - Deriving one or more sub classes from a single base class is called single inheritance.
  - One base class but there can be n number of sub classes derived from it.
  - Example 'Bank' is a single base class from where we derive StateBank, AxisBank as sub classes.

SINGLE INHERITANCE

- **Single Inheritance:**



- It is convention that we should use the arrow head towards the base class i.e the super class in the inheritance diagrams.

SINGLE INHERITANCE

- A Python program showing single inheritance in which two sub classes are derived from a single base class.

```
class Bank(object):
    cash = 100000000
    def available_cash(cls):
        print(cls.cash)

class AxisBank(Bank):
    pass

class StateBank(Bank):
    cash = 20000000
    def available_cash(cls):
        print(cls.cash + Bank.cash)

a = AxisBank()
a.available_cash()

s = StateBank()
s.available_cash()
```

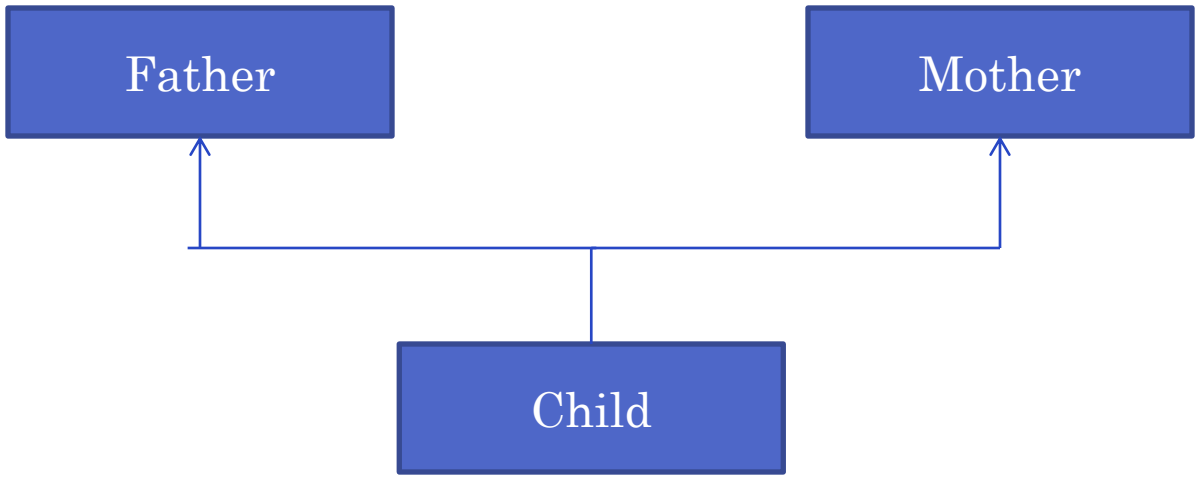
```
In [10]: %run -i "D:/Python Programs/singleinh.py"
100000000
120000000
```

MULTIPLE INHERITANCE

- **Multiple Inheritance:**
  - Deriving sub classes from multiple base classes is called multiple inheritance.
  - There will be more than one super class and there may be one or more sub classes.
  - All the members of the super classes are by default available to sub classes and the sub classes in turn can have their own members.

MULTIPLE INHERITANCE

- **Syntax for multiple inheritance:**
  - **Class Subclass(Baseclass1, Baseclass2,…):**
- Best example for multiple inheritance is that parents, an the children inheriting the qualities of the parents.



- Best example is children inheriting the qualities of the parents.

MULTIPLE INHERITANCE

- Python program to implement multiple inheritance using two base classes.

```
class Father:
    def height(self):
        print('Height is 6.0 foot')

class Mother:
    def color(self):
        print('Color is brown')

class Child(Father,Mother):
    pass

c = Child()
print('Child\'s inherited qualities: ')
c.height()
c.color()
```

```
In [11]: %run -i "D:/Python Programs/multiinh.py"
Child's inherited qualities:
Height is 6.0 foot
Color is brown
```

PROBLEMS IN MULTIPLE INHERITANCE

- If the sub class has a constructor, it overrides the super class constructor and the super class constructor is not available to the sub class.
- In multiple inheritance assume that a sub class 'C' is derived from two super classes 'A'and 'B' having their own constructors.
- Even the sub class 'C ' also has its own constructor.
- To derive C from A and B we write:
  - **class C(A,B):**
- In class C' constructor , we call the super class constructor using super().__init__().
- If we create an object of class C, first the class C constructor is called.
- Then super().__init__() will call class A's constructor.

PROBLEMS IN MULTIPLE INHERITANCE

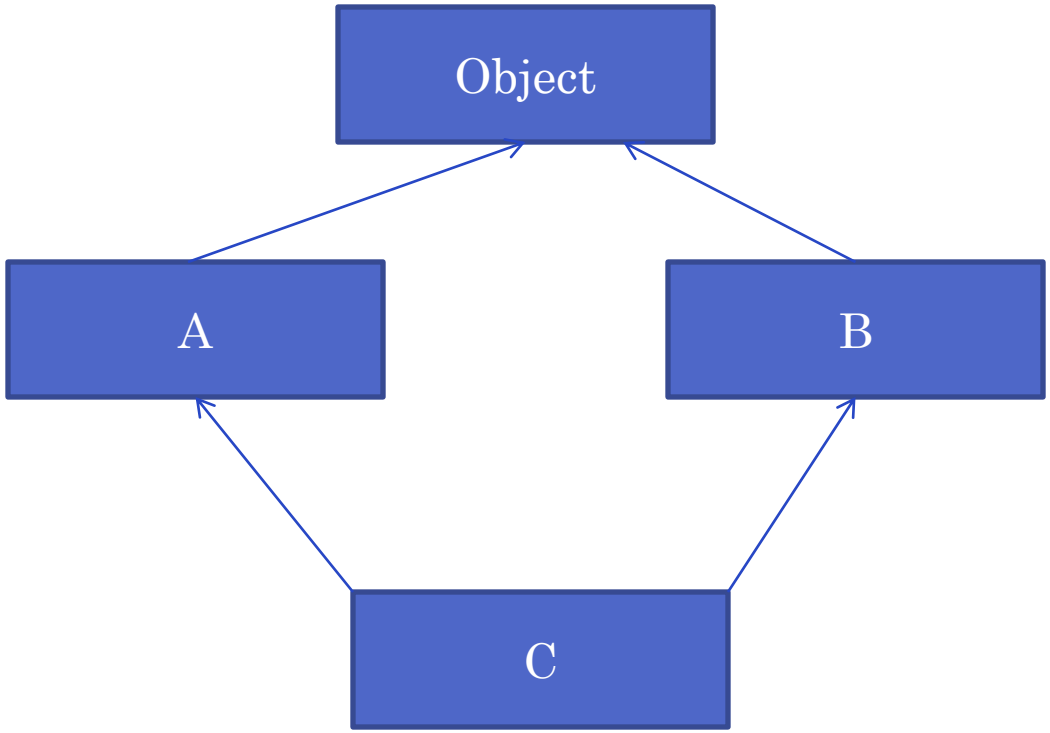
```
class A(object):
    def __init__(self):
        self.a='a'
        print(self.a)
class B(object):
    def __init__(self):
        self.b='b'
        print(self.b)
class C(A,B):
    def __init__(self):
        self.c='c'
        print(self.c)
        super().__init__()
o = C()
```

```
In [2]: %run "c:\users\mdhanl~1\appdata\local\temp\tmpf61zbc.py"
c
a
```

PROBLEMS IN MULTIPLE INHERITANCE

- The output of the program indicates that when class C object is created the C's constructor is called.
- In class C, we used the statement super().__init__() that calls the class A's constructor only.
- We can access only class A's instance variables and not that of class B.
- We created subclass C as class C(A,B):
- This means class C is derived from A and B.
- All classes are sub classes of object class internally, we can take classes A and B are sub classes of object class.

PROBLEMS IN MULTIPLE INHERITANCE



Effect of class C(A,B)

PROBLEMS IN MULTIPLE INHERITANCE

- In the previous figure , class A is at the left hand side and class B is at the right side for class C.
- The searching for any attribute or method will start from the sub class C.
- So, C's constructor is accessed first.
- It will display 'c'.
- The last line in code super().__init__() will call the constructor of the class which is at the left hand side.
- So, class A's constructor is executed and 'a' is displayed.
- If class A does not have constructor , then it will call the constructor at right hand side i.e B.
- Since class A has constructor , the search stopped here.

PROBLEMS IN MULTIPLE INHERITANCE

- If the class C is derived as class C(B,A):
- The output will be:
  - c
  - b
- Problem here is class C is unable to access constructors of both the super classes.
- C cannot access all the instance variables of both of its super classes.
- If C wants to access instance variables of both of its super classes, then solution is to use super().__init__() in every class.

PROBLEMS IN MULTIPLE INHERITANCE

```
class A(object):
    def __init__(self):
        self.a='a'
        print(self.a)
        super().__init__()
class B(object):
    def __init__(self):
        self.b='b'
        print(self.b)
        super().__init__()
class C(A,B):
    def __init__(self):
        self.c='c'
        print(self.c)
        super().__init__()
o = C()
```

```
In [3]: %run "D:/Python Programs/inhproblem1.py"
c
a
b
```

Method resolution order [MRO]

- The search will start from C.
- As the object of C is created, the constructor of C is called and ‘c’ is displayed.
- Then super().__init__() will call the constructor of left side class i.e of A.
- So, the constructor of A is called and ‘a’ is displayed.
- Inside the constructor of A again call its super class constructor using super().__init__().
- Since ‘object’ is the super class for A, an attempt to execute object class constructor will be done.
- But object class does not have any constructor.

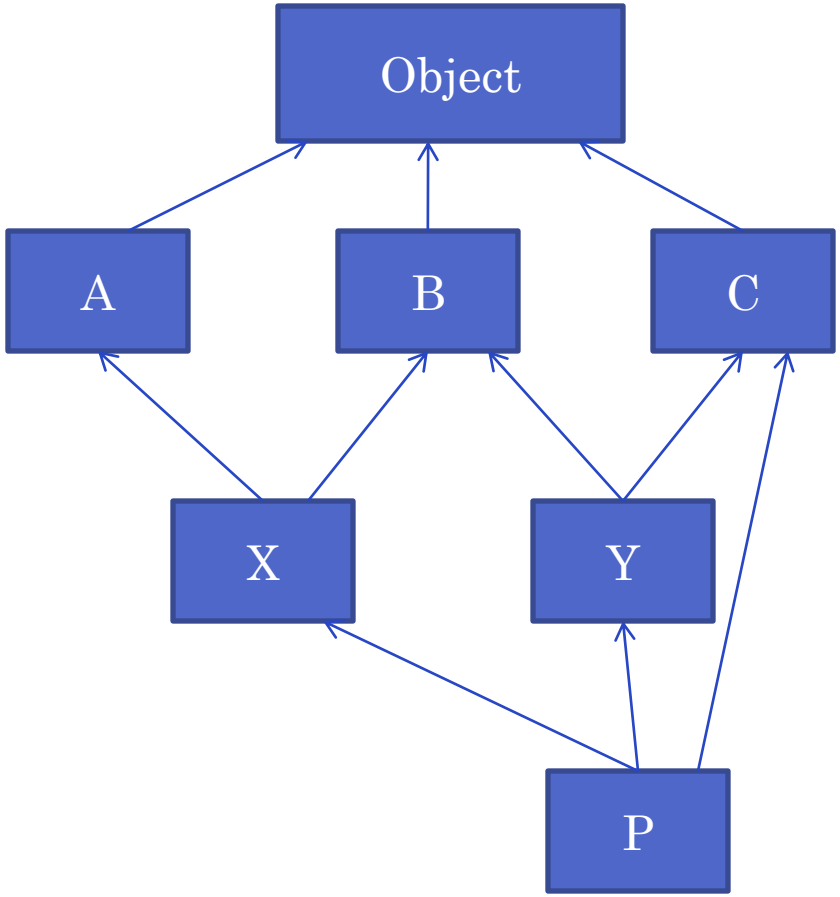
METHOD RESOLUTION ORDER [MRO]

- So, the search will continue down to right hand side class of object class which is class B.
- Hence B's constructor is called and 'b' is displayed.
- The statement super().__init__() will attempt to execute constructor of B's super class which is object class.
- Since object class is already visited the search stops here.
- The output will be 'c','a', and 'b'.
- **Searching in this manner for constructors or methods is called Method Resolution Order [MRO].**

Method resolution order [MRO]

- Three principles followed by MRO:
  - Search for the sub class before going for its base classes. If B is inherited from A, it will search B first and then goes to A.
  - When a class is inherited from several classes, it searches in the order from left to right in the base classes. If class C is inherited from A and B as C(A,B) then it will search in A and then in B.
  - It will not visit an class more than once. A class in inheritance hierarchy is traversed only once exactly.
- To know the MRO, use mro() method as Classname.mro().

Method resolution order [MRO]



Method resolution order [MRO]

```
class A(object):
    def method(self):
        print('A class method')
        super().method()
class B(object):
    def method(self):
        print('B class method')
        super().method()
class C(object):
    def method(self):
        print('C class method')
class X(A,B):
    def method(self):
        print('X class method')
        super().method()
class Y(B,C):
    def method(self):
        print('Y class method')
        super().method()
class P(X,Y,C):
    def method(self):
        print('P class method')
        super().method()
        print(P.mro())

p=P()
p.method()
```

```
In [4]: %run "D:/Python Programs/mro.py"
P class method
X class method
A class method
Y class method
B class method
C class method
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>, <class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class
'object'>]
```

POLYMORPHISM

POLYMORPHISM

- Polymorphism is a word that came from two Greek words, *poly* means many and *morphos* means forms.
- If something exhibits various forms, it is called polymorphism.
- Example wheat flour we can make burgers, rotis or loaves of bread.
- Same wheat flour is taking different edible forms and hence we can say wheat flour is exhibiting polymorphism.

POLYMORPHISM

- A variable may store different types of data, an object may exhibit different behaviors in context or a method may perform various tasks in Python.
- **If a variable, object or method exhibits different behavior in different contexts, it is called polymorphism.**
- Python has built-in polymorphism.
- The following topics are examples of polymorphism in Python:
  - Duck Typing philosophy of Python
  - Operator overloading
  - Method overloading
  - Method overriding

DUCK TYPING PHILOSOPHY OF PYTHON

- In Python, the data type of the variables is not explicitly declared.
- Each variable or object has a type and the type is implicitly assigned depending on the purpose for which the variable is used.
- Example 'x' is a variable.
- If we store integer into that variable its type is taken as 'int' and if string is stored into that variable, its type is taken as 'str'.
- To check the type of variable or object, use type() function.

```
x = 5
print(type(x))
x='Hello'
print(type(x))
```

```
In [2]: %run "c:\users\mdhanl~1\appdata\
<class 'int'>
<class 'str'>
```

DUCK TYPING PHILOSOPHY OF PYTHON

- Python's variables are names that point to memory locations where data is stored.
- They are not worried about which data we are going to serve.
  - Python's type system is 'strong' because every variable or object has a type that we can check with the type() function.
  - Python's type system is 'dynamic' since the type of a variable is not explicitly declared, but it changes with the content being stored.

DUCK TYPING PHILOSOPHY OF PYTHON

- Python program to invoke a method on an object without knowing the type (or class) of the object.

```
class Duck:
    def talk(self):
        print('Quack Quack!')
class Human:
    def talk(self):
        print('Hello, hi!')
def call_talk(obj):
    obj.talk()
x=Duck()
call_talk(x)
x=Human()
call_talk(x)
```

```
In [5]: %run "D:/Python Programs/ducktype.py"
Quack Quack!
Hello, hi!
```

DUCK TYPING PHILOSOPHY OF PYTHON

- From the previous program we understood that we do not need a type in order to invoke an existing method on an object.
- If the method is defined on the object, then it can be called.
- So, when we passed Duck object to call_talk(), it has called the talk() method of Duck class.
- When we passed Human object to call_talk(), it has called the talk() method of Human type.
- During runtime if it is found that the method does not belong to that object, there will be an error called 'AttributeError'.

DUCK TYPING PHILOSOPHY OF PYTHON

- Python program to call a method that does not appear in the object passed to the method.

```
class Dog:
    def bark(self):
        print('Bow,wow!')
class Duck:
    def talk(self):
        print('Quack Quack!')
class Human:
    def talk(self):
        print('Hello, hi!')
def call_talk(obj):
    obj.talk()
x=Duck()
call_talk(x)
x=Human()
call_talk(x)
x=Dog()
call_talk(x)
```

```
D:\Python Programs\ducktype1.py in call_talk(obj)
      9         print('Hello, hi!')
     10 def call_talk(obj):
---> 11     obj.talk()
     12 x=Duck()
     13 call_talk(x)
AttributeError: 'Dog' object has no attribute 'talk'
```

DUCK TYPING PHILOSOPHY OF PYTHON

- In previous program we are passing Dog class object to call_talk() method in the last statement.
- call_talk() method called the talk() method as: obj.talk().
- Since the object type is Dog, this call to talk() method tries to execute talk() method of Dog class which was not found. Hence there was an error.

DUCK TYPING PHILOSOPHY OF PYTHON

- The statement:
  - def call_talk(obj):
    - obj.talk()
- This method is calling talk() method of the object ‘obj’.
- It is not bothered about which class object it is.
- We can pass any class object as long as that object contains the talk() method.
- We can pass Duck object or Human object since they contain the talk() method.
- But when we pass the Dog object, there would be an error since it does not contain talk() method.

DUCK TYPING PHILOSOPHY OF PYTHON

- In Python, we never worry about the type of objects.
- The object type is distinguished only at runtime.
- **If 'it walks like a duck and talks like a duck, it must be a duck' . This is called duck typing.**
- The behavior of the talk() method is changing depending on the object type.
- This is an example for polymorphism of methods.

DUCK TYPING PHILOSOPHY OF PYTHON

- Code to check whether the object has a method or not with the help of hasattr() function.
  - hasattr(object, attribute)

DUCK TYPING PHILOSOPHY OF PYTHON

```
class Dog:
    def bark(self):
        print('Bow,wow!')
class Duck:
    def talk(self):
        print('Quack Quack!')
class Human:
    def talk(self):
        print('Hello, hi!')

def call_talk(obj):
    if hasattr(obj,'talk'):
        obj.talk()
    elif  hasattr(obj,'bark'):
        obj.bark()
    else:
        print('Wrong object passed')

x=Duck()
call_talk(x)
x=Human()
call_talk(x)
x=Dog()
call_talk(x)
```

```
In [3]: %run "D:/Python Programs/ducktype2.py"
Quack Quack!
Hello, hi!
Bow,wow!
```

POLYMORPHISM

- Operator Overloading:
  - An operator is a symbol that performs some action.
  - Example '+' is an operator that performs addition operation when used on numbers.
  - When an operation can perform different actions, it is used to exhibit polymorphism.

```
print(10+15)
s1="New"
s2="York"
print(s1+s2)

a = [10, 20, 30]
b = [5, 15, -10]

print(a+b)
```

```
In [20]: %run -i "D:/Python Programs/opoverload.py"
25
NewYork
[10, 20, 30, 5, 15, -10]
```

POLYMORPHISM

- A Python program to use addition operator to add the contents of two objects.

```
class BookX:
    def __init__(self,pages):
        self.pages = pages
    def __add__(self, other):
        return self.pages+other.pages

class BookY:
    def __init__(self,pages):
        self.pages = pages

b1=BookX(100)
b2=BookY(150)
print('Total pages= ',b1+b2)
```

```
In [23]: %run -i "F:/Python Programs/opoverload.py"
('Total pages= ', 250)
```

POLYMORPHISM

- A Python program to overload the addition operator (+) to make it act on class objects.

```
class BookX:
    def __init__(self,pages):
        self.pages = pages

    def __add__(self,other):
        return self.pages + other.pages

class BookY:
    def __init__(self,pages):
        self.pages = pages

b1=BookX(100)
b2=BookY(15)
print('Total pages=',b1+b2)
```

```
In [1]: %run -i "D:/Python Programs/plusoverload.py"
Total pages= 115
```

POLYMORPHISM

- Program where we want to overload the greater than (>) operator.
- This operator is normally used on numbers to compare them.
- It returns True or False depending on the result.
- If we want to use it on objects, we have to overload it.
- So, the method __gt__() should be overridden.

POLYMORPHISM

- A Python program to overload greater than (>) operator to make it act on class objects.

```
class Ramayan:
    def __init__(self,pages):
        self.pages = pages

    def __gt__(self,other):
        return self.pages>other.pages

class Mahabharat:
    def __init__(self,pages):
        self.pages = pages

b1 = Ramayan(1000)
b2 = Mahabharat(1500)
if(b1>b2):
    print('Ramayan has more pages')
else:
    print('Mahabarat has more pages')
```

```
In [1]: %run "D:/Python Programs/gtoverload.py"
Mahabarat has more pages
```

POLYMORPHISM

- A Python program to overload the multiplication (*) operator to make it act on objects.

```
class Employee:
    def __init__(self,name,salary):
        self.name = name
        self.salary = salary

    def __mul__(self,other):
        return self.salary*other.days

class Attendance:
    def __init__(self,name,days):
        self.name = name
        self.days = days

x1 = Employee('Dhanlaxmi',1000.00)
x2 = Attendance('Dhanlaxmi', 30)
print('This month salary= ',x1*x2)
```

```
In [2]: %run "D:/Python Programs/muloverload.py"
This month salary=  30000.0
```

POLYMORPHISM

- Method Overloading:
  - If a method is written such that it can perform more than one task it is called method overloading.
  - Method overloading in the languages like Java we call a method as:
    - sum(10,15)
    - sum(10,15,20)
  - In first call we pass 2 arguments, and in second call, we are passing three arguments.
  - Sum() method is performing two distinct operations, finding sum of two numbers or sum of three numbers.
  - This is called method overloading.

POLYMORPHISM

- In Java we write two sum() methods with different number of parameters as:
  - Sum(int a, int b) { }
  - Sum(int a, int b, int c) { }
- Since same name is given for these two methods, the user feels that the same method is performing the two operations.
- But method overloading is not available in Python.
- Writing more than one method with same name is not possible in Python.
- We can achieve method overloading by writing same method with several parameters.

POLYMORPHISM

- The method performs the operation depending on the number of arguments passed in the method call.
- We can write a sum() method with default value 'None' for the arguments as:
  - def sum(self, a=None, b=None, c=None):
    - if a!=None and b!=None and c!=None:
      - print('Sum of Three= ' , a+b+c)
    - elif a!=None and b!=None:
      - print('Sum of Two= ' , a+b)

POLYMORPHISM

- Python program to show method overloading to find sum of two or three numbers.

```
class Myclass:
    def sum(self, a=None, b=None, c=None):
        if a!=None and b!=None and c!=None:
            print('Sum of three= ' , a+b+c)
        elif a!=None and b!=None:
            print('Sum of two= ' , a+b)
        else:
            print('Please enter two or three arguments')

m = Myclass()
m.sum(10,15,20)
m.sum(10.5, 20.5)
m.sum(10)
```

```
In [1]: %run -i "D:/Python Programs/methoverload.py"
Sum of three=  45
Sum of two=  31.0
Please enter two or three arguments
```

POLYMORPHISM

- Method Overriding:
  - When there is a method in the super class, writing the same method in the sub class so that it replaces the super class method is called method overriding.
  - The programmer overrides the super class methods when he does not want to use them in sub class.
  - In inheritance, if we create super class object, we can access all the members of the super class but nit the members of the sub class.
  - If we create sub class object, then both the super class and sub class members are available since the sub class object contains a copy of the super class.
  - Hence, in inheritance we always create sub class object.

POLYMORPHISM

- Python program to override the super class method in sub class.

```
import math
class Square:
    def area(self, x):
        print('Square area=%.4f}'% x*x)

class Circle(Square):
    def area(self, x):
        print('Circle area=%.4f'% (math.pi*x*x))

c = Circle()
c.area(15)
```

```
In [2]: %run -i "D:/Python Programs/methoverride.py"
Circle area=706.8583
```