Python Programming

MODULE – IV

Agenda:

- Classes and Object-Oriented Programming (OOP): OOP,
- Classes, Class Attributes
- Instances,Instance Attributes
- Binding and Method Invocation
- Composition
- Subclassing and Derivation
- Inheritance
- Built-in Functions for Classes, Instances, and Other Objects,
- Types vs. Classes/Instances
- Customizing Classes with Special Methods,
- Privacy,
- Delegation and Wrapping

Object Oriented Programming (OOP)

- In all the programs, we have designed our program around functions i.e. blocks of statements which manipulate the data. This is called the procedure-oriented way of programming.
- There is another way of organizing our program which is to combine data and functionality and wrap it inside something called an object. This is called the object oriented programming paradigm.
- Classes and objects are the two main aspects of object oriented programming.
- A **class** creates a new type where **objects** are instances of the class.
- Object Oriented Programming is a way of computer programming using the idea of "objects" to represents data and methods. It is also, an approach used for creating neat and reusable code instead of a redundant one.
- The program is divided into self-contained objects or several mini-programs. Every Individual object represents a different part of the application having its own logic and data to communicate within themselves.
- Now, to get a more clear picture of why we use oops instead of pop, I have listed down the differences below.

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifies	POP does not have any access specifier.	OOP has access specifies named Public, Private, Protected.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: C, VB, FORTRAN, and Pascal……	Examples of OOP are: C++, JAVA, VB.NET, C#.NET,PYTHON….

- Major principles of object-oriented programming system are given below.

  - Class
  - Object
  - Method
  - Inheritance
  - Polymorphism
  - Data Abstraction
  - Encapsulation

Class:

- A Class in Python is a logical grouping of data and functions. It gives the freedom to create data structures that contains arbitrary content and hence easily accessible.
- A class is a "blueprint" or "prototype" to define an object. Every object has its properties and methods. That means a class contains some properties and methods.
- A class is the blueprint from which the individual objects are created. Class is composed of three things: a name, attributes, and operations
- For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax

```
class ClassName:
    <statement-1>
    .
    .
    <statement-N>
```

Object

- An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.
- Object is composed of three things: a name, attributes, and operations or Objects are an instance of a class. It is an entity that has state and behavior.

Syntax: **object_name = ClassName(arguments)**

To define class you need to consider following points

Step 1) In Python, classes are defined by the "Class" keyword

```
class myClass():
```

Step 2) Inside classes, you can define functions or methods that are part of this class

```
def ds(self):
  print ("DS Branch")
def cse (self,value):
    print ("CSE Branch" ,value)
```

Here we have defined ds that prints "DS Branch"

Another method we have defined is cse that prints "CSE Branch"+ value . value is the variable supplied by the calling method

Step 3) Everything in a class is indented, just like the code in the function, loop, if statement, etc. Anything not indented is not in the class

Example:

```
class myClass:
   def ds(self):
      print("DS Branch")
   def cse(self,value):
      print("CSE Branch",value)
```

"self" in Python:

- The self-argument refers to the object itself. Hence the use of the word self. So inside this method, self will refer to the specific instance of this object that's being operated on.
- Self is the name preferred by convention by Pythons to indicate the first parameter of instance methods in Python. It is part of the Python syntax to access members of objects

Step 4) To make an object of the class

```
c = myClass()
```

Step 5) To call a method in a class

```
c.ds()
c.cse(5)
```

- Notice that when we call the ds or cse, we don't have to supply the self-keyword. That's automatically handled for us by the Python runtime.
- Python runtime will pass "self" value when you call an instance method on in instance, whether you provide it deliberately or not You just have to care about the non-self arguments

Step 6) Here is the complete code

```
# Example file for working with classes
class myClass:
   def ds(self):
      print("DS Branch")
   def cse(self,value):
      print("CSE Branch",value)
c=myClass()
c.ds()
c.cse(5)
```

output: DS Branch

CSE Branch 5

Python Constructor:

- Python Constructor in object-oriented programming with Python is a special kind of method/function we use to initialize instance members of that class.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be executed only once.
- Constructor can take atleast one argument(atleast self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor

Types of Constructors:

We observe three types of Python Constructors, two of which are in our hands. Let's begin with the one that isn't.

- Default Constructor
- Non- Parameterized Constructor
- Parameterized Constructor

Default Constructor:

A constructor that Python lends us when we forget to include one. This one does absolutely nothing but instantiates the object; it is an empty constructor- without a body.

Example:

```
class defaultConstructor:
    def display(self):
        print("This is Default Constructor")
dc=defaultConstructor()
dc.display()
```

OutPut:

```
This is Default Constructor
```

Non- Parameterized Constructor:

- When we want a constructor to do something but none of that is to manipulate values, we can use a non-parameterized constructor.
- As we know that a constructor always has a name init and the name init is prefixed and suffixed with a double underscore(__). We declare a constructor using def keyword, just like methods.

Syntax:

```
def __init__(self):
    # body of the constructor
```

Example:

```
class NonParameterizedConstructor:
   def __init__(self):
      print("Non-Parameterized Constructor")
   def display(self,name):
      print("Name=",name)
npc=NonParameterizedConstructor()
npc.display('Raj')
```

OutPut:

Non- Parameterized Constructor
Name= Raj

Parameterized constructor:

- Constructor with parameters is known as parameterized constructor.The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example:

```
class Addition:
        first = 0
        second = 0
        answer = 0
        def __init__(self, f, s):
                self.first = f
                self.second = s

        def display(self):
                print("First number = " + str(self.first))
                print("Second number = " + str(self.second))
                print("Addition of two numbers = " + str(self.answer))

        def calculate(self):
                self.answer = self.first + self.second
obj = Addition(1000, 2000)
obj.calculate()
obj.display()
```

OutPut:

```
First number = 1000
Second number = 2000
Addition of two numbers = 3000
```

Class Attributes and Instance Attributes

Attributes are noting but variables we the following types are there in python.

Types of Class Variables in Python: There are three different types of variables in OOPs in python.

- Instance variables (object level variables)
- Static variables (class level variables)
- Local variables

Instance Variables in Python:

If the value of a variable is changing from object to object then such variables are called as instance variables.

```
class Student:
  def __init__(self, name, id):
     self.name=name
     self.id=id
s1=Student('Srav', 1)
s2=Student('Raj', 2)
print("Studen1 info:")
print("Name: ", s1.name)
print("Id : ", s1.id)
print("Studen2 info:")
print("Name: ",s2.name)
print("Id : ",s2.id)
```

OutPut:

```
Studen1 info:
Name:  Srav
Id :  1
Studen2 info:
Name:  Raj
Id :  2
```

Static variables in Python:

- If the value of a variable is not changing from object to object, such types of variables are called static variables or class level variables. We can access static variables either by class name or by object name. Accessing static variables with class names is highly recommended than object names.

Example:

```
class Student:
  college='MREC'
  def __init__(self, name, id):
```

```
        self.name=name
        self.id=id
    s1=Student('SRAV', 1)
    s2=Student('RAJ', 2)
    print("Studen1 info:")
    print("Name: ", s1.name)
    print("Id : ", s1.id)
    print("College name n : ", Student.college)
    print("\n")
    print("Studen2 info:")
    print("Name: ",s2.name)
    print("Id : ",s2.id)
    print("College name : ", Student.college)
```

OutPut:

```
Studen1 info:
Name:  SRAV
Id :  1
College name n :  MREC

Studen2 info:
Name:  RAJ
Id :  2
College name :  MREC
```

Local Variables in Python:

- The variable which we declare inside of the method is called a local variable. Generally, for temporary usage we create local variables to use within the methods. The scope of these variables is limited to the method in which they are declared. They are not accessible out side of the methods.

Example:

```
class mrec:
   dept="DS"#static variable
   def display(self):
     dept="CSE" #Local Variable
     print(dept)
d=mrec()
d.display()
print(d.dept)
```

OutPut:

```
CSE
DS
```

Binding and Method Invocation:

- There are three main types of methods in Python.
  - Instance methods
  - Static methods
  - Class methods.

Instance methods:

- Instance methods are the most common type of methods in Python classes. These are so called because they can access unique data of their instance.
- And we call it as default method in python.
- If you have two objects each created from a car class, then they each may have different properties. They may have different colors, engine sizes, seats, and so on.
- Instance methods are methods which act upon the instance variables of the class. They are bound with instances or objects, that"s why called as instance methods. The first parameter for instance methods should be self variable which refers to instance. Along with the self variable it can contain other variables as well.
- Any method you create will automatically be created as an instance method, unless you tell Python otherwise.

Example:

```
class Test:
   def __init__(self, a, b):
      self.a = a
      self.b = b
   def avg(self):
      return (self.a + self.b) / 2
s1 = Test(10, 20)
print( s1.avg() )
```

OutPut:

```
15.0
```

Class Methods:

- Class methods are methods which act upon the class variables or static variables of the class. We can go for class methods when we are using only class variables (static variables) within the method.
- Class methods should be declared with @classmethod.
- Just as instance methods have 'self' as the default first variable, class method should have 'cls' as the first variable. Along with the cls variable it can contain other variables as well.
- Class methods are rarely used in python

Example:

```
class Mrec:
  Dept="CSE"
  def Dept_name(self,name):
     print("Instance method=",name)
  @classmethod
  def get_Dept(cls):
     return cls.Dept
m=Mrec()
m.Dept_name("DS")
print("Class method=",Mrec.get_Dept())
```

OutPut:

```
Instance method= DS
Class method= CSE
```

Static methods

- A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.
- Inside these methods we won't use any instance or class variables. No arguments like cls or self are required at the time of declaration.
- We can declare static method explicitly by using @staticmethod decorator.
- We can access static methods by using class name or object reference

Example:

```
class Demo:
  @staticmethod
  def sum(x, y):
     print(x+y)
  @staticmethod
  def multiply(x, y):
     print(x*y)
Demo.sum(2, 3)
Demo.multiply(2,4)
```

OutPut:

```
5
8
```

Example:

```
class Demo:
   x=10
   y=5
   def __init__(self,x,y):
      self.x=x
      self.y=y
   def add(self):
      print("Sum=",self.x+self.y)
   @classmethod
   def sub(cls):
      print("Sub=", cls.x-cls.y)
   @staticmethod
   def multiply(x,y):
     print("Mul=",x*y)
d=Demo(10,5)
d.add()
Demo.sub()
Demo.multiply(10,5)
```

OutPut:

```
Sum= 15
Sub= 5
Mul= 50
```

Inheritance or Is-A Relation in Python

- The inheritance is the process of acquiring the properties of one class to another class.
- Inheritance in python programming is the concept of deriving a new class from an existing class.
- Using the concept of inheritance we can inherit the properties of the existing class to our new class.
- The new derived class is called the **child class** and the existing class is called the **parent class**.
- The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.
- The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass.**

Advantages of Inheritance:

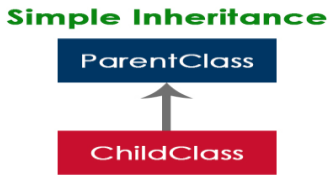
- **Code reusability**- we do not have to write the same code again and again, we can

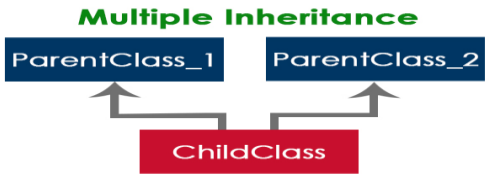
just inherit the properties we need in a child class.

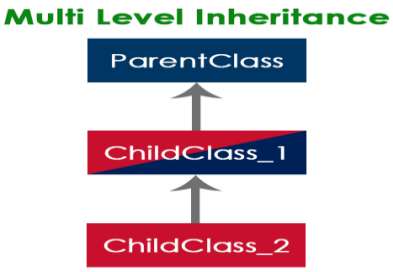
- It represents a real world relationship between parent class and child class.
- It is transitive in nature. If a child class inherits properties from a parent class, then all other sub-classes of the child class will also inherit the properties of the parent class.

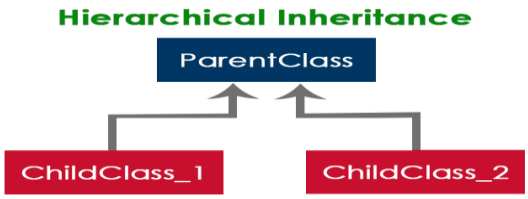
- There are five types of inheritances, and they are as follows.
  - Simple Inheritance (or) Single Inheritance
  - Multiple Inheritance
  - Multi-Level Inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance

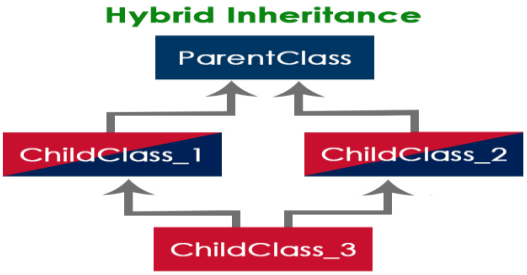
The following picture illustrates how various inheritances are implemented.











Creating a Child Class

- In Python, we use the following general structure to create a child class from a parent class.

Syntax:

```
class ChildClassName(ParentClassName):
    ChildClass implementation
```

Simple Inheritance (or) Single Inheritance

In this type of inheritance, one child class derives from one parent class. Look at the following example code.

Example

```
class Parent:
        def func1(self):
                print("This function is in parent class.")

class Child(Parent):
        def func2(self):
                print("This function is in child class.")

object = Child()
object.func1()
object.func2()
```

OutPut:

```
This function is in parent class.
This function is in child class.
```

Multi-Level Inheritance

- In this type of inheritance, the child class derives from a class which already derived from another class. Look at the following example code.

Example:

```
class Parent:
    def func1(self):
        print('this is function 1')
class Child(Parent):
    def func2(self):
        print('this is function 2')
class Child2(Child):
    def func3(self):
        print('this is function 3')
ob = Child2()
ob.func1()
ob.func2()
ob.func3()
```

OutPut:

```
this is function 1
this is function 2
this is function 3
```

Hierarchical inheritance:

- When we derive or inherit more than one child class from one (same) parent class. Then this type of inheritance is called hierarchical inheritance.

Example:

```
class Parent:
        def func1(self):
                print("This function is in parent class.")
class Child1(Parent):
        def func2(self):
                print("This function is in child 1.")
class Child2(Parent):
        def func3(self):
                print("This function is in child 2.")
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

OutPut:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Multiple Inheritance:

When child class is derived or inherited from the more than one parent classes. This is called multiple inheritance. In multiple inheritance, we have two parent classes/base classes and one child class that inherits both parent classes' properties.

Example:

```
class Father:
        fathername = ""
        def father(self):
                print(self.fathername)
class Mother:
        mothername = ""
        def mother(self):
                print(self.mothername)

class Son(Mother, Father):
```

```
        def parents(self):
            print("Father :", self.fathername)
            print("Mother :", self.mothername)
    s1 = Son()
    s1.fathername = "Raj"
    s1.mothername = "Srav"
    s1.parents()
```

OutPut:

```
Father : Raj
Mother : Srav
```

Hybrid Inheritance:

Hybrid inheritance satisfies more than one form of inheritance ie. It may be consists of all types of inheritance that we have done above. It is not wrong if we say Hybrid Inheritance is the combinations of simple, multiple, multilevel and hierarchical inheritance. This type of inheritance is very helpful if we want to use concepts of inheritance without any limitations according to our requirements.

Example:

```
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.")
object = Student3()
object.func1()
object.func2()
object.func4()
```

OutPut:

```
This function is in school.
This function is in student 1.
This function is in student 3.
```

Super() Function in Python:

- super() is a predefined function in python. By using super() function in child class, we can call,
  - Super class constructor.
  - Super class variables.
  - Super class methods.

1. Calling super class constructor from child class constructor using super()

Example:

```
class A:
  def __init__(self):
     print("super class A constructor")
class B(A):
  def __init__(self):
     print("Child class B constructor")
     super().__init__()
     b=B()
```

OutPut:

```
Child class B constructor
super class A constructor
```

2. Calling super class method from child class method using super()

```
class A:
  def m1(self):
     print("Super class A: m1 method")
class B(A):
  def m1(self):
     print("Child class B: m1 method")
     super().m1()
b=B()
b.m1()
```

Output:

```
Child class B: m1 method
Super class A: m1 method
```

3. Calling super class variable from child class method using super()

```
class A:
  x=10
  def m1(self):
     print("Super class A: m1 method")
```

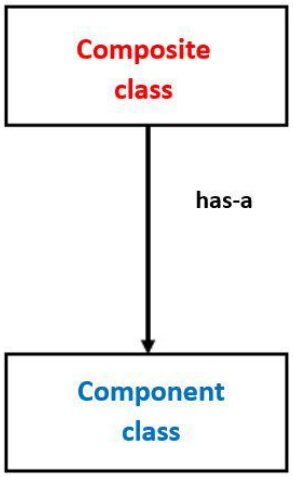
```
class B(A):
  x=20
  def m1(self):
     print('Child class x variable', self.x)
     print('Super class x variable', super().x)
b=B()
b.m1()
```

Output:

```
Child class x variable 20
Super class x variable 10
```

Composition (Has A Relation):

- It is one of the fundamental concepts of Object-Oriented Programming. In this concept, we will describe a class that references to one or more objects of other classes as an Instance variable. Here, by using the class name or by creating the object we can access the members of one class inside another class. It enables creating complex types by combining objects of different classes. It means that a class Composite can contain an object of another class Component. This type of relationship is known as **Has-A Relation**.
- In composition one of the classes is composed of one or more instance of other classes. In other words one class is container and other class is content and if you delete the container object then all of its contents objects are also deleted.



Syntax:

class A :

```
        # variables of class A
        # methods of class A
        …
        …
    class B :
        # by using "object" we can access member's of class A.
        object = A()

        # variables of class B
        # methods of class B
        …
        …
```

Example:

```
class Component:
        def __init__(self):
                print('Component class object created…')
        def m1(self):
                print('Component class m1() method executed…')
class Composite:
        def __init__(self):
                self.obj1 = Component()
                print('Composite class object also created…')
        def m2(self):
                print('Composite class m2() method executed…')
                self.obj1.m1()
obj2 = Composite()
obj2.m2()
```

OutPut:

```
Component class object created…
Composite class object also created…
Composite class m2() method executed…
Component class m1() method executed…
```

Privacy or Python Access Modifiers:

- In most of the object-oriented languages access modifiers are used to limit the access to the variables and functions of a class. Most of the languages use three types of access modifiers, they are –
  - Private
  - Public

- Protected.

- Just like any other object oriented programming language, access to variables or functions can also be limited in python using the access modifiers. Python makes the use of underscores to specify the access modifier for a specific data member and member function in a class.
- Access modifiers play an important role to protect the data from unauthorized access as well as protecting it from getting manipulated.
- When inheritance is implemented there is a huge risk for the data to get destroyed(manipulated) due to transfer of unwanted data from the parent class to the child class. Therefore, it is very important to provide the right access modifiers for different data members and member functions depending upon the requirements.

Python: Types of Access Modifiers

- There are 3 types of access modifiers for a class in Python. These access modifiers define how the members of the class can be accessed. Of course, any member of a class is accessible inside any member function of that same class. Moving ahead to the type of access modifiers, they are:

Access Modifier: Public

- The members declared as Public are accessible from outside the Class through an object of the class.

Example:

```
class Student:
   def __init__(self,name,dept):
      self.name=name#Public attribute
      self.dept=dept#Public attribute
class Stud(Student):
   pass
s1=Student("Raj","CSE")
print("Name=",s1.name)
print("Dept=",s1.dept)
d=Stud("Srav","DS")
print("Name=",d.name)
print("Dept=",d.dept)
```

OutPut:

```
Name=Raj
Dept=CSE
Name=Srav
Dept=DS
```

Access Modifier: Private

- These members are only accessible from within the class. No outside Access is allowed.

- It is also not possible to inherit the private members of any class (parent class) to derived class (child class). Any instance variable in a class followed by self keyword and the variable name starting with double underscore ie. self.__varName are the private accessed member of a class.

Example:

```
class Student:
   def __init__(self, name, dept):
      self.__name = name # private
      self.__dept  = dept  # private
s1=Student("Raj","CSE")
print("Name=",s1.__name)
print("Dept=",s1.__dept)
```

OutPut: error

protected Access Modifier:

- Protected variables or we can say protected members of a class are restricted to be used only by the member functions and class members of the same class. And also it can be accessed or inherited by its derived class (child class).
- We can modify the values of protected variables of a class. The syntax we follow to make any variable protected is to write variable name followed by a single underscore (_) ie. _varName.

Example:

```
class Student:
   def __init__(self, name, dept):
      self._name = name # Protected
      self._dept  = dept  #Protected
class Stu(Student):
   pass
s1=Student("Raj","CSE")
print("Name=",s1._name)
print("Dept=",s1._dept)
s2=Stu("Srav","DS")
print("Name=",s2._name)
print("Dept=",s2._dept)
```

OutPut:

```
Name= Raj
Dept= CSE
Name= Srav
Dept= DS
```

Polymorphism in Python

- Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types. This makes programming easier.
- Polymorphism means having vivid or different forms. In the programming world, Polymorphism refers to the ability of the function with the same name to carry different functionality altogether.

Types of Polymorphism :

- **Compile time Polymorphism**
- **Run time Polymorphism**

Compile time Polymorphism or Method Overloading:

- Unlike many other popular object-oriented programming languages such as Java, Python doesn't support compile-time polymorphism or method overloading. If a class or Python script has multiple methods with the same name, the method defined in the last will override the earlier one.
- Python doesn't use function arguments for method signature, that's why method overloading is not supported in Python.

Example:

```
class OverloadDemo:
  def multiply(self,a,b):
     print(a*b)
  def multiply(self,a,b,c):
     print(a*b*c)
m=OverloadDemo()
m.multiply(5,10)
```

OutPut:

```
Traceback (most recent call last):
 File "F:\R20-python\lab\ac.py", line 7, in <module>
   m.multiply(5,10)
TypeError: multiply() missing 1 required positional argument: 'c'
```

Run time Polymorphism or Method Overriding:

- In Python, whenever a method having same name and arguments is used in both derived class as well as in base or super class then the method used in derived class is said to override the method described in base class. Whenever the overridden method is called, it always invokes the method defined in derived class. The method used in base class gets hidden.

Example:

```
class methodOverride1:
   def display(self):
      print("method invoked from base class")

class methodOverride2(methodOverride1):
   def display(self):
      print("method invoked from derived class")

ob=methodOverride2()
ob.display()
```

OutPut:

```
method invoked from derived class
```

Built in Functions for Classes, Instances, and Other Objects,

In Python we have different typed of built in functions in Python.

1. **hasattr() Function**

- The python hasattr() function returns true if an object has given named attribute. Otherwise, it returns false.

Syntax: hasattr(object, attribute)

Parameters

- **object:** It is an object whose named attribute is to be checked.
- **attribute:** It is the name of the attribute that you want to search.
- **Return:**It returns true if an object has given named attribute. Otherwise, it returns false.

Example:

```
class Demo:
   name="raj"
   dept="CSE"
obj=Demo()
print(hasattr(Demo,'name'))
print(hasattr(Demo,'rollno'))
```

OutPut:

```
True
False
```

2. **getattr() Function**

- The python getattr() function returns the value of a named attribute of an object. If it is not found, it returns the default value.

Syntax: getattr(object, attribute, default)

Parameters:

- **object:** An object whose named attribute value is to be returned.
- **attribute:** Name of the attribute of which you want to get the value.
- **default (optional):** It is the value to return if the named attribute does not found.

Return: It returns the value of a named attribute of an object. If it is not found, it returns the default value.

Example:

```
class Demo:
   name="raj"
   dept="CSE"
obj=Demo()
print("name=",getattr(Demo,'name'))
print("college=",getattr(Demo,'college',"MREC"))
```

OutPut:

```
name= raj
college= MREC
```

3. setattr() Function

- Python setattr() function is used to set a value to the object's attribute. It takes three arguments an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it. The signature of the function is given below.

Syntax: setattr (object, name, value)

Parameters

- **object:** It is an object which allows its attributes to be changed.
- **name :** A name of the attribute.
- **value :** A value, set to the attribute.

Return: It returns None to the caller function.

Example:

```
class Demo:
   name=""
   dept=""
   id=0
   def __init__(self,name,dept,id):
      self.name=name
      self.dept=dept
      self.id=id
obj=Demo("Raj","CSE",1)
print(obj.name)
print(obj.dept)
print(obj.id)
```

```
setattr(obj,'college','MREC')
print(obj.college)
```

OutPut:

```
Raj
CSE
1
MREC
```

4. delattr() Function

- Python delattr() function is used to delete an attribute from a class. It takes two parameters first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

Syntax: delattr (object, name)

Parameters

- **object:** Object of the class which contains the attribute.
- **name:** The name of the attribute to delete. It must be a string.

Return: It returns a complex number.

Example:

```
class Demo:
   name=""
   dept=""
   id=0
   def __init__(self,name,dept,id):
      self.name=name
      self.dept=dept
      self.id=id
obj=Demo("Raj","CSE",1)
print(obj.name)
print(obj.dept)
print(obj.id)
setattr(obj,'college','MREC')
print(obj.college)
delattr(obj,'college')
print(obj.college)
```

OutPut:

```
Raj
CSE
1
MREC
Traceback (most recent call last):
 File "F:\R20-python\lab\ac.py", line 16, in <module>
```

```
print(obj.college)
AttributeError: 'Demo' object has no attribute 'college'
```

5. **isinstance() Function**
   - Python isinstance() function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns True. Otherwise returns False. It also returns true if the class is a subclass.
   - The isinstance() function takes two arguments object and classinfo and returns either True or False. The signature of the function is given below.

Syntax: isinstance(object, classinfo)

Parameters

- **object:** It is an object of string, int, float, long or custom type.
- **classinfo:** Class name.

Return: It returns boolean either True or False.

6. **issubclass() function**
   - The issubclass() function returns True if the specified object is a subclass of the specified object, otherwise False.

Syntax: issubclass(object, subclass)

Parameter

- **object** It is an object of string, int, float, long or custom type.
- **subclass** Name of the subclass

Return: It returns boolean either True or False.

Example:

```
class A:
  pass
class B(A):
  pass
b=B()
print("b is an instance of the class B:",isinstance(b,B))
print("B is sub class of A Class:",issubclass(B,A))
```

OutPut:

```
b is an instance of the class B: True
B is sub class of A Class: True
```

Types vs. Classes/Instances:

Properties	Types	Class
Origin	Pre-defined data types	User-defined data types
Stored structure	Stored in a stack	Reference variable is stored in stack and the original object is

		stored in heap
When copied	Two different variables is created along with different assignment even though the both variables having the same address if they are pointing the same value	Two reference variable is created but both are pointing to the same object on the heap
When changes are made in the copied variable	Change does not reflect in the original ones.	Changes reflected in the original ones.
Default value	Primitive datatypes having the default value like 0 for int 0.0 for float etc.	No default value for the reference variable which is created for a class

Example	a=15
	print("Type of a=",type(a))
	
	output:
	Type of a= <class 'int'>
		class A:
		def __init__(self,a):
		self.a=a
		obj=A(10)
		print("Type of obj=",type(obj))
		
		output:
		Type of obj= <class '__main__.A'>

Delegation and Wrapping

- Delegation is the mechanism through which an actor assigns a task or part of a task to another actor. This is not new in computer science, as any program can be split into blocks and each block generally depends on the previous ones. Furthermore, code can be isolated in libraries and reused in different parts of a program, implementing this "task assignment". In an OO system the assignee is not just the code of a function, but a full-fledged object, another actor.
- The main concept to retain here is that the reason behind delegation is code reuse. We want to avoid code repetition, as it is often the source of regressions; fixing a bug

in one of the repetitions doesn't automatically fix it in all of them, so keeping one single version of each algorithm is paramount to ensure the consistency of a system.

- Delegation helps us to keep our actors small and specialised, which makes the whole architecture more flexible and easier to maintain (if properly implemented). Changing a very big subsystem to satisfy a new requirement might affect other parts system in bad ways, so the smaller the subsystems the better (up to a certain point, where we incur in the opposite problem, but this shall be discussed in another post).

Example:

```
class Dept:
   def __init__(self,insem,endsem):
      self.insem=insem
      self.endsem=endsem
   def marks(self):
      return self.insem+self.endsem

class student:
   def __init__(self,sname,year,insem,endsem):
      self.sname=sname
      self.year=year
      self.obj_Dept=Dept(insem,endsem)
   def tmarks(self):
      print("The Total Marks of %s"
%self.sname,self.obj_Dept.marks())
s=student('Raj','First',20,65)
s.tmarks()
```

OutPut:

```
The Total Marks of Raj 85
```