Chapter 5.

Object Oriented Programming in Python

12 Marks

Introduction:

- Python follows ***object oriented programming paradigm***. It deals with declaring python classes and objects which lays the foundation of OOP's concepts.
- Python programming offers OOP style programming and provides an easy way to develop programs. It uses the OOP concepts that makes python ***more powerful to help design a program that represents real world entities***.
- Python supports OOP concepts such as ***Inheritance, Method Overriding, Data abstraction and Data hiding***.

Important terms in OOP/ Terminology of OOP-

(Different OOP features supported by Python)

- **Class**- Classes are defined by the user. The class ***provides the basic structure for an object***. It ***consists of data members and method members*** that are used by the instances(object) of the class.
- **Object**- A unique ***instance of a data structure*** that is defined by its class. An object ***comprises both data members and methods***. Class itself does nothing but real functionality is achieved through their objects.

- **Data Member**: A variable ***defined in either a class or an object***; it holds the data associated with the class or object.
- **Instance variable**: A variable that is ***defined in a method***, its scope is only within the object that defines it.
- **Class variable**: A variable that is ***defined in the class*** and can be used by all the instance of that class.
- **Instance:** An object is an instance of a class.
- **Method**: They are ***functions that are defined in the definition of class*** and are used by various instances of the class.
- **Function Overloading**: A function ***defined more than one time*** with different behavior. (different arguments)
- **Encapsulation**: It is the ***process of binding together the methods and data variables as a single entity*** i.e. class. It hides the data within the class and makes it available only through the methods.
- **Inheritance**: The ***transfer of characteristics of a class to other classes*** that are derived from it.
- **Polymorphism**: It allows ***one interface to be used for a set of actions***. It means same function name(but different signatures) being used for different types.
- **Data abstraction**: It is the ***process of hiding the implementation details and showing only functionality*** to the user.

Classes-

- Python is OOP language. Almost everything in python is an object with its properties and methods.
- Object is simply a collection of data(variables) and methods(functions) that acts on those data.

Creating Classes:

A class is a block of statement that combine data and operations,

which are performed on the data, into a group as a single unit and

acts as a blueprint for the creation of objects.

Syntax:

class ClassName:

‘ Optional class documentation string

#list of python class variables

Python class constructor

#Python class method definitions

- In a class we can define variables, functions etc. While writing ***function in class we have to pass atleast one argument*** that is called ***self parameter***.
- The self parameter is a reference to the class itself and is used to access variables that belongs to the class.

Example: Creating class in .py file

```
class student:
    def display(self):  # defining method in class
        print("Hello Python")
```

- In python programming **self is a default variable** that contains the ***memory address of the instance of the current class.***
- So we can use self to refer to all the instance variable and instance methods.

Objects and Creating Objects-

- An object is an instance of a class that has some attributes and behavior.
- Objects can be used to access the attributes of the class.

Example:

```
class student:
    def display(self):  # defining method in class
        print("Hello Python")
s1=student()     #creating object of class
s1.display()     #calling method of class using object
```

Output:

```
Hello Python
```

Example: Class with get and put method

```
class car:
    def get(self,color,style):
        self.color=color
        self.style=style
    def put(self):
        print(self.color)
        print(self.style)
c=car()
c.get('Brio','Red')
c.put()
```

Output*:*

```
Brio
Red
```

Instance variable and Class variable:

- **Instance variable** is ***defined in a method and its scope is only within the object*** that defines it.
- Every object of the class has its own copy of that variable. Any changes made to the variable don’t reflect in other objects of that class.

- **Class variable** is ***defined in the class*** and can be ***used by all the instances of that class***.
- ***Instance variables are unique for each instance***, while ***class variables are shared by all instances***.

Example: For instance and class variables

```
class sample:
    x=2       # x is class variable
    def get(self,y):  # y is instance variable
        self.y=y
s1=sample()
s1.get(3)    # Access attributes
print(s1.x," ",s1.y)
s2=sample()
s2.y=4
print(s2.x," ",s2.y)
```

Output:

```
2   3
2   4
```

Data Hiding:

- Data hiding is ***a software development technique specifically used in object oriented programming to hide internal object details***(data members).
- It ***ensures exclusive data access to class members*** and ***protects object integrity by preventing unintended or intended changes***.
- Data hiding is also known as **information hiding**. An objects attributes may or may not be visible outside the class definition.
- We need to **name attributes with a double underscore**(_ _) **prefix** and ***those attributes the are not directly visible to outsiders***. Any variable prefix with double underscore is called **private variable** which is accessible only with class where it is declared.

Example: For data hiding

```
class counter:
    __secretcount=0    # private variable
    def count(self):  # public method
        self __secretcount+=1
        print("count= ",self.__secretcount)  # accessible in the same class
c1=counter()
c1.count() # invoke method
c1.count()
print("Total count= ",c1.__secretcount) # cannot access private variable directly
```

Output:

```
count=  1
count=  2
```

Traceback (most recent call last):

File "D:\python programs\class_method.py", line 9, in <module>

print("Total count= ",c1.__secretcount) # cannot access private variable directly

AttributeError: 'counter' object has no attribute '__secretcount'

Data Encapsulation and Data Abstraction:

- We can ***restrict access of methods and variables in a class with the help of encapsulation***. It will prevent the data being modified by accident.
- **Encapsulation** ***is used to hide the value or state of a structured data object inside a class***, preventing unauthorized parties direct access to them.
- **Data abstraction** refers to ***providing only essential information about the data to the outside world, hiding the background details of implementation***.
- **Encapsulation** is a ***process to bind data and functions together into a single unit i.e. class*** while **abstraction** is a process in which the ***data inside the class is the hidden from outside world***.
- In short hiding internal details and showing functionality is known as **abstraction**.
- To support encapsulation, declare the methods or variables as private in the class. The ***private methods cannot be called by the object directly***. It can be called only from within the class in which they are defined.
- Any ***function with double underscore is called private method***.

Access modifiers for variables and methods are:

- **Public methods / variables-** Accessible from anywhere inside the class, in the sub class, in same script file as well as outside the script file.

- **Private methods / variables**- Accessible only in their own class. Starts with two underscores.

Example: For access modifiers with data abstraction

```
class student:
    __a=10 #private variable
    b=20 #public variable
    def __private_method(self): #private method
        print("Private method is called")
    def public_method(self): #public method
        print("public method is called")
        print("a= ",self.__a) #can be accessible in same class
s1=student()
# print("a= ",s1.__a) #generate error
print("b=",s1.b)
# s1.__private_method() #generate error
s1.public_method()
```

Output:

```
b= 20
public method is called
a=  10
```

Creating Constructor:

- Constructors are generally used for ***instantiating an object***.
- The task of constructors is to ***initialize***(assign values) to the data members of the class when an object of class is created.
- In Python the **__init__()** method is called the constructor and is always called when an object is created.

***Syntax of constructor declaration* :**

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

Example: ***For creating constructor use_ _init_ _ method called as constructor***.

```
class student:
    def __init__(self,rollno,name,age):
        self.rollno=rollno
        self.name=name
        self.age=age
        print("student object is created")
p1=student(11,"Ajay",20)
print("Roll No of student= ",p1.rollno)
print("Name No of student= ",p1.name)
print("Age No of student= ",p1.age)
```

***Output**:*

student object is created

Roll No of student= 11

Name No of student= Ajay

Age No of student= 20

Programs:

Define a class rectangle using length and width.It has a method which can compute area.

```
class rectangle:
    def __init__(self,L,W):
        self.L=L
        self.W=W
    def area(self):
        return self.L*self.W
r=rectangle(2,10)
print(r.area())
```

Output

20

Create a circle class and initialize it with radius. Make two methods getarea and getcircumference inside this class

```
class circle:
    def __init__(self,radius):
        self.radius=radius
    def getarea(self):
        return 3.14*self.radius*self.radius
    def getcircumference(self):
        return 2*3.14*self.radius
c=circle(5)
print("Area=",c.getarea())
print("Circumference=",c.getcircumference())
```

Output:

```
Area= 78.5
Circumference= 31.400000000000002
```

Types of Constructor:

There are two types of constructor- ***Default constructor and Parameterized constructor***.

Default constructor- The default constructor is simple constructor which does not accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

Example: **Display Hello message using Default constructor(It does not accept argument)**

```
class student:
    def __init__(self):
        print("This is non parameterized constructor")
    def show(self,name):
        print("Hello",name)
s1=student()
s1.show("World")
```

Output:

```
This is non parameterized constructor
Hello World
```

Example: **Counting the number of objects of a class**

```
class student:
    count=0
    def __init__(self):
        student.count=student.count+1
```

```
s1=student()

s2=student()

print("The number of student objects",student.count)
```

Output:

The number of student objects 2

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: For parameterized constructor

```
class student:
    def __init__(self,name):
        print("This is parameterized constructor")
        self.name=name
    def show(self):
        print("Hello",self.name)
s1=student("World")
```

s1.show()

Output:

This is parameterized constructor

Hello World

Destructor:

A class can define a special method called destructor with the help of **_ _del_ _()**. It is invoked automatically when the instance (object) is about to be destroyed.

It is mostly ***used to clean up non memory resources*** used by an instance(object).

Example: For Destructor

```
class student:
    def __init__(self):
        print("This is non parameterized constructor")

    def        ____del____(self):
        print("Destructor called")
s1=student()
s2=student()
del s1
```

Output:

This is non parameterized constructor

This is non parameterized constructor

Destructor called

Method Overloading:

- ***Method overloading is the ability to define the method with the same name but with a different number of arguments and data types***.
- With this ability one method can perform different tasks, depending on the number of arguments or the types of the arguments given.
- Method overloading is a concept in which a method in a class performs operations according to the parameters passed to it.
- As in other language we can write a program having two methods with same name but with different number of arguments or order of arguments but in python if we will try to do the same we get the following issue with method overloading in python.

  Example-

```
# To calculate area of rectangle
def area(length,breadth):
    calc=length*breadth
    print(calc)
# To calculate area of square
def area(size):
    calc=size*size
```

```
    print(calc)
area(3)
area(4,5)
```

Output-

```
9
Traceback (most recent call last):
  File "D:\python programs\trial.py", line 10, in <module>
    area(4,5)
TypeError: area() takes 1 positional argument but 2 were given
```

- Python does not support method overloading i.e it is not possible to define more than one method with the same name in a class in python.
- This is because method arguments in python do not have a type. A method accepting one argument can be called with an integer value, a string or a double as shown in example.

Example-

```
class demo:
    def print_r(self,a,b):
        print(a)
        print(b)
obj=demo()
obj.print_r(10,'S')
```

```
obj.print_r('S',10)
```

Output:

```
10
S
S
10
```

- In the above example same method works for two different data types.
- It is clear that method overloading is not supported in python but that does not mean that we cannot call a method with different number of arguments. There are couple of alternatives available in python that make it possible to call the same method but with different number of arguments.

Using Default Arguments:

It is possible to provide default values to method arguments while defining a method. If method arguments are supplied default values, then it is not mandatory to supply those arguments while calling method as shown in example.

***Example 1*: Method overloading with deafult arguments**

```
class demo:
    def arguments(self,a=None,b=None,c=None):
        if(a!=None and b!=None and c!=None):
            print("3 arguments")
        elif (a!=None and b!=None):
```

```
            print("2 arguments")
        elif a!=None:
            print("1 argument")
        else:
            print("0 arguments")
obj=demo()
obj.arguments("Amol","Kedar","Sanjay")
obj.arguments("Amit","Rahul")
obj.arguments("Sidharth")
obj.arguments()
```

Output-

```
3 arguments
2 arguments
1 argument
0 arguments
```

***Example 2*: With a method to perform different operations using method overloading**

```
class operation:
    def add(self,a,b):
        return a+b
op=operation()
# To add two integer numbers
print("Addition of integer numbers= ",op.add(10,20))
# To add two floating numbers
print("Addition of integer numbers= ",op.add(11.12,12.13))
# To add two strings
print("Addition of stings= ",op.add("Hello","World"))
```

Output-

```
Addition of integer numbers=  30
Addition of integer numbers=  23.25
Addition of stings=  HelloWorld
```

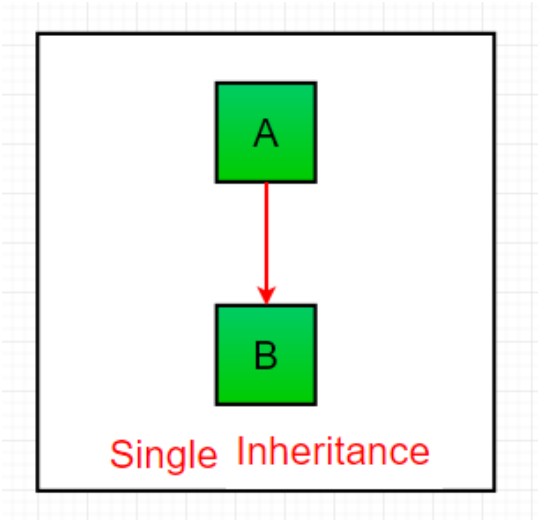
Inheritance:

The mechanism of designing and constructing classes from other classes is called inheritance.

Inheritance is the capability of one class to derive or inherit the properties from some another class.

The new class is called **derived class** or **child class** and the class from which this derived class has been inherited is the **base class** or **parent class**. The benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.



Syntax:

```
Class A:
        # Properties of class A
Class B(A):
        # Class B inheriting property of class A
        #  more properties of class B
```

Example 1: **Example of Inheritance without using constructor**

```
class vehicle:  #parent class
    name="Maruti"
    def display(self):
        print("Name= ",self.name)
class category(vehicle): # drived class
    price=400000
    def disp_price(self):
        print("price= ",self.price)
car1=category()
car1.display()
car1.disp_price()
```

Output:

Name= Maruti

price= 400000

Example 2: **Example of Inheritance using constructor**

```
class vehicle:  #parent class
    def __init__(self,name,price):
        self.name=name
        self.price=price
    def display(self):
        print("Name= ",self.name)
class category(vehicle): # drived class
    def __init__(self,name,price):
        vehicle.__init__(self,name,price) #pass data to base constructor
    def disp_price(self):
        print("price= ",self.price)
car1=category("Maruti",400000)
car1.display()
car1.disp_price()
```

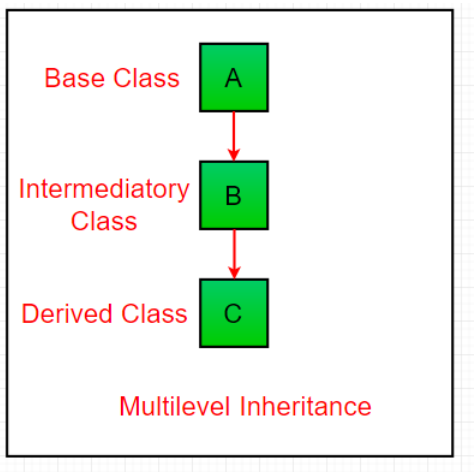
```
car2=category("Honda",600000)

car2.display()

car2.disp_price()
```

Output:

```
Name=  Maruti

price=  400000

Name=  Honda

price= 600000
```

Multilevel Inheritance:

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



Syntax:

```
Class A:
        # Properties of class A
Class B(A):
        # Class B inheriting property of class A
        #  more properties of class B
Class C(B):
        # Class C inheriting property of class B
        # thus, Class C also inherits properties of class A
        # more properties of class C
```

Example 1: Python program to demonstrate multilevel inheritance

```
#Mutilevel Inheritance
class c1:
    def display1(self):
        print("class c1")
class c2(c1):
    def display2(self):
        print("class c2")
class c3(c2):
```

```
    def display3(self):
        print("class c3")
s1=c3()
s1.display3()
s1.display2()
s1.display1()
```

Output:

```
class c3
class c2
class c1
```

Example 2: Python program to demonstrate multilevel inheritance

```
# Base class
class Grandfather:
    grandfathername =""
    def grandfather(self):
        print(self.grandfathername)
```

```
# Intermediate class
class Father(Grandfather):
    fathername = ""
    def father(self):
        print(self.fathername)


# Derived class
class Son(Father):
    def parent(self):
        print("GrandFather :", self.grandfathername)
        print("Father :", self.fathername)


# Driver's code
s1 = Son()
s1.grandfathername = "Srinivas"
s1.fathername = "Ankush"
s1.parent()
```

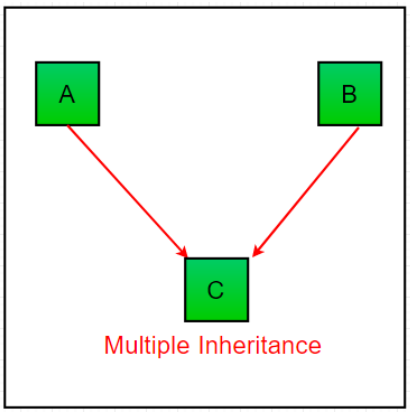
Output:

```
GrandFather : Srinivas
```

Father : Ankush

Multiple Inheritance:

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



Syntax:

Class A:

    # variable of class A

    # functions of class A

Class B:

    # variable of class B

    # functions of class B

Class C(A,B):

```
        # Class C inheriting property of both class A and B
        # more properties of class C
```

Example: **Python program to demonstrate multiple inheritance**

```
# Base class1
class Father:
    def display1(self):
        print("Father")
# Base class2
class Mother:
    def display2(self):
        print("Mother")
# Derived class
class Son(Father,Mother):
    def display3(self):
        print("Son")
s1 = Son()
s1.display3()
s1.display2()
s1.display1()
```

Output:

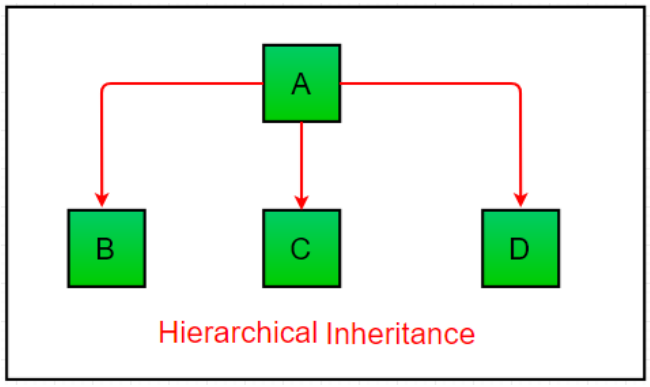
Son

Mother

Father

Hierarchical Inheritance:

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



***Example* : Python program to demonstrate Hierarchical inheritance**

```
# Base class

class Parent:
```

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


# Derived class1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")


# Derived class2
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.

Method Overriding:

Method overriding is an ability of a class to change the implementation of a method provided by one of its base class. Method overriding is thus a strict part of inheritance mechanism.

To override a method in base class, we must define a new method with sam name and same parameters in the derived class.

Overriding is a very important part of OOP since it is feature that makes inheritance exploit its full power. Through method overriding a class may "copy" another class, avoiding duplicated code and at the same time enhance or customize part of it.

Example: For method overriding

```
class A:
    def display(self):
        print("This is base class")
```

```
class B(A):
    def display(self):
        print("This is derived class")
obj=B()              # instance of child
obj.display()        # child class overriden method
```

Output-

```
This is derived class
```

Using super() Method:

The super() method gives you access to methods in a super class from the subclass that inherits from it.

The super() method returns a temporary object of the superclass that then allows you to call that superclass's method.

Example: For method overriding with super()

```
class A:
    def display(self):
        print("This is base class")
class B(A):
    def display(self):
        super().display()
```

```
    print("This is derived class")
obj=B()              # instance of child
obj.display()        # child class overriden method
```

Output-

```
This is base class
This is derived class
```

Composition Classes:

- In composition we ***do not inherit from the base class*** but ***establish relationship between classes through the use of instance variables*** that are references to other objects.
- Composition means that ***an object knows another object and explicitly delegates some tasks to it***. While ***inheritance is implicit***, ***composition is explicit*** in python.
- We use composition when we want ***to use some aspects of another class without promising all of the features of that other class***.

Syntax:

```
Class GenericClass:
        Define some attributes and methods
```

```
Class AspecificClass:
        Instance_variable_of_generic_class=GenericClass
        #use this instance somewhere in the class
        Some_method(instance_varable_of_generic_class)
```

- For example, we have three classes email, gmail and yahoo. In email class we are referring the gmail and using the concept of composition.

Example:

```
class gmail:
    def send_email(self,msg):
        print("sending '{}' from gmail".format(msg))
class yahoo:
    def send_email(self,msg):
        print("sending '{}' from yahoo".format(msg))
class email:
    provider=gmail()
    def set_provider(self,provider):
        self.provider=provider
    def send_email(self,msg):
```

```
        self.provider.send_email(msg)

client1=email()

client1.send_email("Hello")

client1.set_provider(yahoo())

client1.send_email("Hello")
```

Output:

```
sending 'Hello' from gmail

sending 'Hello' from yahoo
```

Customization via Inheritance specializing inherited methods:

- The ***tree-searching model of inheritance*** turns out to be a great way to specialize systems. Because ***inheritance finds names in subclasses before it checks superclasses***, subclasses can replace default behavior by redefining the superclass's attributes.
- In fact, you can build entire systems as hierarchies of classes, which are extended by adding new external subclasses rather than changing existing logic in place.
- ***The idea of redefining inherited names leads to a variety of specialization techniques***.

- For instance, ***subclasses may replace inherited attributes completely, provide attributes that a superclass expects to find, and extend superclass methods by calling back to the superclass from an overridden method***.

***Example*- For specilaized inherited methods**

```
class A:
    "parent class"  #parent class
    def display(self):
        print("This is base class")
class B(A):
    "Child class"    #derived class
    def display(self):
        A.display(self)
        print("This is derived class")
obj=B()   #instance of child
obj.display()   #child calls overridden method
```

Output:

```
This is base class
This is derived class
```

- In the above example derived class.display() just extends base class.display() behavior rather than replacing it completely.
- ***Extension is the only way to interface with a superclass***.
- The following program defines multiple classes that illustrate a variety of common techniques.

Super

Defines a method function and a delegate that expects an action in a

subclass

Inheritor

Doesn't provide any new names, so it gets everything defined in Super

Replacer

Overrides Super's method with a version of its own

Extender

Customizes Super's method by overriding and calling back to run the

default

Provider

Implements the action method expected by Super's delegate method

***Example*- Various ways to customize a common superclass**

```
class super:
    def method(self):
        print("in super.method")  #default behavior
    def delegate(self):
        self.action()      #expected to be defined
class inheritor(super):
    pass
class replacer(super):  #replace method completely
    def method(self):
        print("in replacer.method")
class extender(super):    #extend method behavior
    def method(self):
        super.method(self)
        print("in extender.method")
class provider(super):   # fill in a required method
    def action(self):
        print("in provider.action“)
for klass in (inheritor,replacer,extender):
```

```
    print("\n"+klass.__name__+"...")
    klass().method()
    print("\n provider...")
x=provider()
x.delegate()
```

Output:

```
inheritor...
in super.method
provider...
replacer...
in replacer.method
provider...
extender...
in super.method
in extender.method
 provider...
in provider.action
```

- When we call the delegate method through provider instance, two independent inheritance searches occur:
- On the initial x.delegate call, Python finds the delegate method in Super, by searching at the provider instance and above. The instance x is passed into the method's self argument as usual.
- Inside the super.delegate method, self.action invokes a new, independent inheritance search at self and above. Because self references a provider instance, the action method is located in the provider subclass.

6.1