Unit 5 **Marks: 12**

Object Oriented Programming in Python

1) Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach. In Python we can easily create and use classes and objects.

2) Python follows object oriented programming paradigm. It deals with declaring python classes and objects which lays the foundation of OOP's concepts.

3) 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.

4) An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects.

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

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

Class:

1) a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated.

2) 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.

3) a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated.

4) Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called instantiation.

Creating Class:

1) The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. 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.

2) 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.

3) Syntax:

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

4) **Example**

```
class Employee:
    id = 10
    name = "Pratyush"
    def display (self):
        print(self.id,self.name)
```

Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.

5) 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.

6) The self parameter is a reference to the class itself and is used to access variables that belongs to the class.

7) The class also contains a function **display()**, which is used to display the information of the **Employee.**

The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

<object-name> = <**class**-name>(<arguments>)

The following example creates the instance of the class Employee defined in the above example.

Example

```
class Employee:
    id = 21
    name = "Neer"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))

# Creating a emp instance of Employee class
emp = Employee()
emp.display()
```

Output:

```
ID: 10
Name: Neer
```

In the above code, we have created the Employee class which has two attributes named id and name and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **emp.** By using it, we can access the attributes of the class.

Object:

1) The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

2) 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.

3) Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code.

4) When we define a class, it needs to create an object to allocate the memory. Consider the following example.

Example:

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year
    def display(self):
        print(self.modelname,self.year)

c1 = car("Toyota", 2022)
c1.display()
```

Output:

```
Toyota 2022
```

In the above example, we have created the class named car, and it has two attributes modelname and year. We have created a c1 object to access the class attribute. The c1 object will allocate memory for these values.

Creating Objects:

1) An object is an instance of a class that has some attributes and behavior.

2) Objects can be used to access the attributes of the class.

3) 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
```

_init_Method:

1) Python suppprts a very unique method as _init_() method.
2) This method gets executed when instane of a class gets created(called automatically).
3) The _init_ method is the python equivalent of the C++ constructor in an object oriented approach.
4) The _init_method is called every time an object is created from class.
5) The _init_method lets the class initialize the objects attributes & serves no other Purpose. It is only used within class.

Example:

```
class Student:
  def  _init_(self, rollno, name):
    self.id = id
    self.name = name
s1 =  Student(7, “Reyansh”)
print(s1.rollno)
print(s1.name)
```

Method

1) The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.
2) They are functions that are defined in the definition of class and are used by various instances of the class.
3) for example, list object have methods called append, insert, sort, remove.
4) The method is implicitly used for an object for which it is called.
5) the method is accessible to data that is contained within the class.

Example:

```
class Student:
    def  _init_(self, rollno, name):
        self.id = id
        self.name = name
    def mymethod(self):
        print("My name is: " +self.name)
s1 =  Student(7, "Reyansh")
s1.mymethod()
```

Data Hiding:

1) Data hiding is a part of object-oriented programming, which is generally used to hide the data information from the user.

2) It includes internal object details such as data members, internal working. It maintained the data integrity and restricted access to the class member.

3) The main working of data hiding is that it combines the data and functions into a single unit to conceal data within a class. We cannot directly access the data from outside the class.

4) Python is the most popular programming language as it applies in every technical domain and has a straightforward syntax and vast libraries

5) We can perform data hiding in Python using the __ double underscore before prefix. This makes the class members private and inaccessible to the other classes.

Advantages of Data Hiding

- The class objects are disconnected from the irrelevant data.
- It enhances the security against hackers that are unable to access important data.
- It isolates object as the basic concept of OOP.
- It helps programmer from incorrect linking to the corrupt data.
- We can isolate the object from the basic concept of OOP.
- It provides the high security which stops damage to violate data by hiding it from the public.

Data Abstraction:

1) Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

2) Abstraction is used to hide internal details and show only functionalities.

3) Data Abstraction is nothing but providing only essential information about the data to the Outside, hiding the background details or implementation.

4) Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

5) It is the process of hiding the implementation details and showing only functionality to the user.

```
class product(ABC):                    // Abstract Class
                                       // Normal Method
def item_list(self, rate):
                                         // Method Definition
print("amount submitted : ",rate)
@abstractmethod
def product(self,rate):                  // Abstract Method
// Method Definition pass
```

Example -

```
# Python program demonstrate
# abstract base class work
from abc import ABC, abstractmethod
class Car(ABC):
```

```
    def mileage(self):
        pass

class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
     def mileage(self):
          print("The mileage is 24kmph ")

class Renault(Car):
    def mileage(self):
           print("The mileage is 27kmph ")

# Driver code
t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

Output:

```
The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph
```

Inheritance

1) Inheritance in Python is *a* mechanism in which one object acquires all the properties and behaviors of a parent *object*.

2) Inheritance allows us to define a class that inherit all the methods and Properties from another class.

3) Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance.

4) It specifies that the child object acquires all the properties and behaviors of the parent object.

5) By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

6) It provides the re-usability of the code.

Syntax

```
class derived-class(base class):
   <class-suite>
```

A class can inherit multiple classes by mentioning all of them inside the bracket.

Syntax

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
   <class - suite>
```

Example 1

```
class Animal:
   def speak(self):
      print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
   def bark(self):
      print("dog barking")
d = Dog()
d.bark()
d.speak()
```

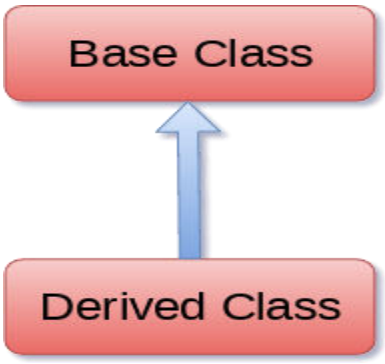
Types of Inheritance:

1) Single Inheritance:

1) in Single inheritance there are one base class and one derived class.

2) derived class inherits the properties of base class.

3)



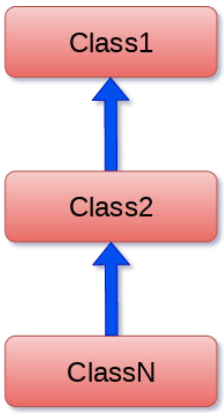
4) Example:

```
# Python program for demonstrating single inheritance

# Here, we will create the base class or the Parent class

class Parent1:
    def func_1(self):
        print ("This function is defined inside the parent class.")
# now, we will create the Derived class
class Child1(Parent1):
    def func_2(self):
        print ("This function is defined inside the child class.")
# Creating object of derived class
object = Child1()
object.func_1()
object.func_2()
```

2) Multi-Level inheritance :

1) Multi-Level inheritance is possible in python like other object-oriented languages.

2) Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



3) The syntax of multi-level inheritance is given below.

Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

Example

```
class Animal:
    def speak(self):

        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
```

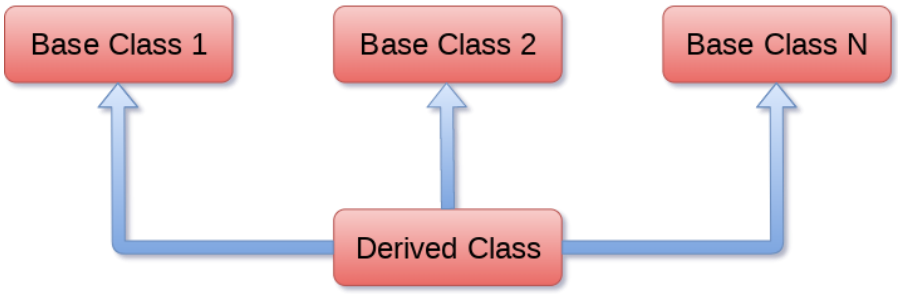
```
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

3) Multiple inheritance:

1) Python provides us the flexibility to inherit multiple base classes and one derived class.



2) Derived class inherits the properties of multiple base classes.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

Polymorphism :

1) Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape.

2) By polymorphism, we understand that one task can be performed in different ways.

3) For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Method Overloading:

1) Method overloading means having two methods with the same name. We can't have two methods with the same name in Python and and we don't need to.

2) In Python, method overloading is not possible; if you want to access the same function with different features, it is better to go for method overriding.

3) In Python you can define a method in such a way that there are multiple ways to call it. Depending on the function definition, it can be called with zero, one, two or more parameters. This is known as method overloading.

4) Overloading is the ability of a function or an operator to behave in different ways based on the parameters that are passed to the function, or the operands that the operator acts on.

5) Overloading a method fosters reusability. For example, instead of writing multiple methods that differ only slightly, we can write one method and overload it.

6) Overloading also improves code clarity and eliminates complexity.

```
class Person:
def Hello(self, name=None):
if name is not None:
print('Hello ' + name)
else:
print('Hello ')
                        # Create instance
obj = Person()
                       # Call the method
obj.Hello()
                        # Call the method with a parameter
obj.Hello('Neer')
```

Output:

Hello
Hello Neer

Method Overriding:

1) We can provide some specific implementation of the parent class method in our child class.

2) When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.

3) We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

4) The overriding method allows a child class to provide a specific implementation of a method that is already provided by one of its parent classes.

5) Inheritance should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.

6) The function that is redefined in the child class should have the same signature as in the parent class i.e. same **number of parameters**.

Example:

```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

Output:

```
Barking
```

Real Life Example of method overriding

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()

print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

Encapsulation

1) Encapsulation is also an essential aspect of object-oriented programming.

2) It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

3) Inheritance should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.

4) The function that is redefined in the child class should have the same signature as in the parent class i.e. same **number of parameters**.

```
# parent class
class Parent:
                                # some random function
    def anything(self):
        print('Function defined in parent class!')

                                  # child class
class Child(Parent):
                                 # empty class definition
    pass


obj2 = Child()
obj2.anything()
```

Object-oriented vs. Procedure-oriented Programming languages

Index	Object-oriented Programming	Procedural Programming
1.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
2.	It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
3.	It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
4.	It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
5.	Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

Python Constructor

1) A constructor is a special type of method (function) which is used to initialize the instance members of the class.

2) In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python

1) In Python, the method the **__init__()** simulates the constructor of the class.

2) This method is called when the class is instantiated.

3) It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

4) We can pass any number of arguments at the time of creating the class object, depending upon the **__init__()** definition.

5) It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

Example

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

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))

emp1 = Employee("ABC", 101)
emp2 = Employee("XYZ", 102)

# accessing display() method to print employee 1 information

emp1.display()

# accessing display() method to print employee 2 information
emp2.display()
```

Output:

```
ID: 101
Name: ABC
ID: 102
Name: XYZ
```

Counting the number of objects of a class

The constructor is called automatically when we create the object of the class. Consider the following example.

Example

```
class Student:
    count = 0
    def __init__(self):
        Student.count = Student.count + 1
s1=Student()
s2=Student()
s3=Student()
print("The number of students:",Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

Consider the following example.

Example

```
class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
```

```
stud = Student()
stud.show("Pratyush")
```

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**.

Consider the following example.

Example

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
stud = Student("Pratyush")
stud.show()
```

Output:

```
This is parametrized constructor
Hello Pratyush
```

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects.

Consider the following example.

Ex.

```
class Student:
    roll_num = 101
    name = "Neer"
     def display(self):
        print(self.roll_num,self.name)

st = Student()
st.display()
```

Python built-in class functions

The built-in functions defined in the class are described in the following table.

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Example

```
class Student:
    def __init__(self, name, id, age):
        self.name = name
        self.id = id
        self.age = age

    # creates the object of the class Student
s = Student("ABC", 101, 22)

# prints the attribute name of the object s
print(getattr(s, 'name'))

# reset the value of attribute age to 23
setattr(s, "age", 23)

# prints the modified value of age
print(getattr(s, 'age'))


# prints true if the student contains the attribute with name id
```

```
print(hasattr(s, 'id'))
# deletes the attribute age
delattr(s, 'age')

# this will give an error since the attribute age has been deleted
print(s.age)
```

Output:

```
ABC
23
True
AttributeError: 'Student' object has no attribute 'age'
```

Built-in class attributes

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

Example

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age
    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
s = Student("John",101,22)
print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

Output:

```
None
{'name': 'John', 'id': 101, 'age': 22}
__main__
```

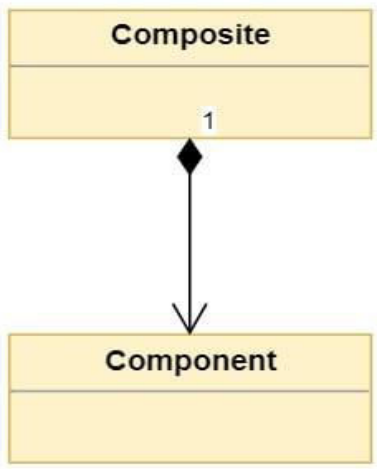
Composition of classes:

1) Composition is a concept that models a **has a** relationship.

2) It enables creating complex types by combining objects of other types.

3) This means that a class Composite can contain an object of another class Component. This relationship means that a Composite **has a** Component.

UML represents composition as follows:



4) Composition is represented through a line with a diamond at the composite class pointing to the component class.

5) The composite side can express the cardinality of the relationship. The cardinality indicates the number or valid range of Component instances the Composite class will contain.

6) Composition is an object oriented design concept that models a **has a** relationship.

7) In composition, a class known as **composite** contains an object of another class known to as **component**. In other words, a composite class **has a** component of another class.

8) Composition allows composite classes to reuse the implementation of the components it contains. The composite class doesn't inherit the component class interface, but it can leverage its implementation.

9) The composition relation between two classes is considered loosely coupled. That means that changes to the component class rarely affect the composite class, and changes to the composite class never affect the component class.

10) This provides better adaptability to change and allows applications to introduce new requirements without affecting existing code.

11) When looking at two competing software designs, one based on inheritance and another based on composition, the composition solution usually is the most flexible. You can now look at how composition works.

You've already used composition in our examples. If you look at the Employee class, you'll see that it contains two attributes:

1. **id** to identify an employee.
2. **name** to contain the name of the employee.

These two attributes are objects that the Employee class has. Therefore, you can say that an Employee **has an** id and **has a** name.

Another attribute for an Employee might be an Address:

```
# In contacts.py

class Address:
    def __init__(self, street, city, state, zipcode, street2=''):
        self.street = street
        self.street2 = street2
        self.city = city
        self.state = state
        self.zipcode = zipcode

    def __str__(self):
        lines = [self.street]
        if self.street2:
            lines.append(self.street2)
        lines.append(f'{self.city}, {self.state} {self.zipcode}')
        return '\n'.join(lines)
```

Customization via Inheritance Specializing Inherited Methods:

1) Create number of objects with the help of classes. These classes also have special ability

   Of creating subclasses. It is also called as inheritance.

2) Instance object generated from a class inherit class attribute.

3) in python there are two important facts. First objects inherits from classes.

4) second classes inherit from their parent class also known as parent class or super class.

5) parent classes are mentioned in round brackets in class header.

6) classes inherits attribute from parent classes.

7) Instance inherit attribute from all accessible classes.

8) each object.attribute reference invokes a new, independent search.

9) logic changes are made by subclassing.