Experiment 06

Write python programs to understand

6.1) Classes, Objects, Constructors, Inner class and Static method,class variables

6.2) Different types of Inheritance

6.3) Polymorphism using Operator overloading, Method overloading, Method overriding, Abstract class, Abstract method and Interfaces in Python.

Roll No.	01
Name	Aamir Ansari
Class	D10-A
Subject	Python Lab
LO Mapped	LO1: Understand the structure, syntax, and semantics of the Python language LO3:illustrate the concepts of object-oriented programming as used in Python

Aim: To understand Object Oriented Programming concepts in python

Introduction:

Class

A class is a blueprint for the object.
We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

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.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, obj is an object of class Parrot.
Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Creating Class and Object in Python

```
class Parrot:
    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output:

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the __init__ method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

We can access the class attribute using __class__.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

Constructors

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
```

Types of constructors :

1. default constructor: The default constructor is a simple constructor which doesn't accept any arguments.It's definition has only one argument which is a reference to the instance being constructed.
2. parameterized constructor: constructor with parameters is known as parameterized constructor.The parameterized constructor takes 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 of constructor

```
class Parrot:

    # default constructor
    def __init__(self):
        self.speak = "Hello"

    # a method for printing data members
    def say(self):
        print(self.speak)


# creating object of the class
obj = Parrot()

# calling the instance method using the object obj
obj.say()
```

Output :

```
Hello
```

Inner Class

A class defined in another class is known as inner class or nested class. If an object is created using child class means inner class then the object can also be used by parent class or root class. A parent class can have one or more inner class but generally inner classes are avoided.

We can make our code even more object oriented by using inner class. A single object of the class can hold multiple sub-objects. We can use multiple sub-objects to give a good structure to our program.

Example –
First we create a class and then the constructor of the class.
After creating a class, we will create another class within that class, the class inside another class will be called as inner class.

```
# create a Color class
class Color:
    # constructor method
    def __init__(self):
        # object attributes
        self.name = 'Green'
        self.lg = self.Lightgreen()

    def show(self):
        print("Name:", self.name)

    # create Lightgreen class
    class Lightgreen:
        def __init__(self):
            self.name = 'Light Green'
            self.code = '024avc'

        def display(self):
            print("Name:", self.name)
            print("Code:", self.code)

# create Color class object
outer = Color()

# method calling
outer.show()

# create a Lightgreen
# inner class object
g = outer.lg

# inner class method calling
g.display()
```

Output:

Green
Name:Green

Light Green
023gfd

Name: Light Green
Code: 023gfd

Static Method

1. Static methods, much like class methods, are methods that are bound to a class rather than its object.
2. They do not require a class instance creation. So, they are not dependent on the state of the object.
3. The difference between a static method and a class method is:
    a. Static method knows nothing about the class and just deals with the parameters.
    b. Class method works with the class since its parameter is always the class itself.

They can be called both by the class and its object.

```
Class.staticmethodFunc()
        or even
Class().staticmethodFunc()
```

Example of creating a static method using staticmethod()

```
class Mathematics:
        def addNumbers(x, y):
                return x + y
# create addNumbers static method
Mathematics.addNumbers = staticmethod(Mathematics.addNumbers)

print('The sum is:', Mathematics.addNumbers(5, 10))
```

Output

```
The sum is: 15
```

Class variable

1. A variable that is shared by all instances of a class.
2. Class variables are defined within a class but outside any of the class's methods.
3. Class variables are not used as frequently as instance variables are.

Inheritance

Inheritance is defined as the capability of one class to derive or inherit the properties from some other class and use it whenever needed. Inheritance provides the following properties:

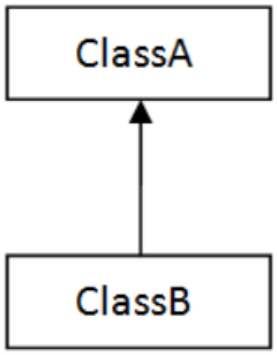
1. It represents real-world relationships well.
2. It provides reusability of code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A,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

Types of inheritance

Types of Inheritance depends upon the number of child and parent classes involved. There are four types of inheritance in python

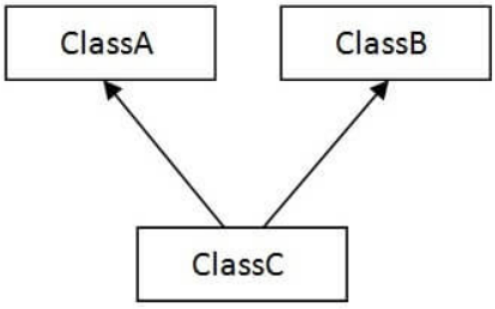
1. Single inheritance:

   Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code



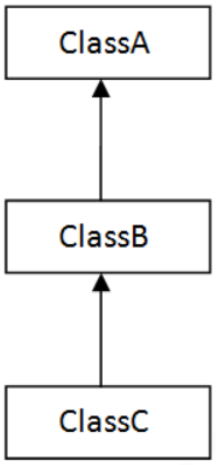
2. Multiple inheritance:

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



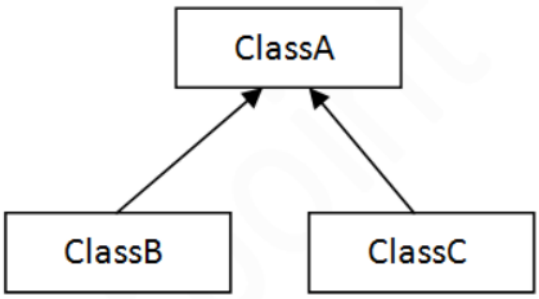
3. Multilevel inheritance:

   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



4. Hierarchical Inheritance:

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



Operator Overloading

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading

```
# Python program to show use of
# + operator for different purposes.

print(10 + 2)

# concatenate two strings
print("Hello "+"World")

# Product two numbers
print(5 * 4)

# Repeat the String
print("HO"*4)
```

Output:

```
12
Hello world
20
HOHOHOHO
```

Method Overloading

Like other languages (for example, method overloading in C++) do, python does not support method overloading by default. But there are different ways to achieve method overloading in Python.
The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

```
# First product method.
# Takes two argument and print their
# product
def product(a, b):
        p = a * b
        print(p)

# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
        p = a * b*c
        print(p)
```

```
# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

Output:

```
100
```

In the above code, we have defined two product method, but we can only use the second product method, as python does not support method overloading. We may define many methods of the same name and different arguments, but we can only use the latest defined method. Calling the other method will produce an error. Like here, calling product(4, 5) will produce an error as the latest defined product method takes three arguments.

To overcome this problem, we have two solutions

1. We can use the arguments to make the same function work differently i.e. as per the arguments
2. By Using Multiple Dispatch Decorator

Method overriding

Method overriding is a concept of object oriented programming that allows us to change the implementation of a function in the child class that is defined in the parent class. It is the ability of a child class to change the implementation of any method which is already provided by one of its parent class(ancestors).

Following conditions must be met for overriding a function:

1. Inheritance should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.
2. 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:

```
# parent class
class Animal:
  # properties
        multicellular = True
        # Eukaryotic means Cells with Nucleus
        eukaryotic = True
```

```
        # function breath
        def breathe(self):
            print("I breathe oxygen.")

    # function feed
        def feed(self):
            print("I eat food.")

# child class
class Herbivorous(Animal):

    # function feed
        def feed(self):
            print("I eat only plants. I am vegetarian.")

herbi = Herbivorous()
herbi.feed()
# calling some other function
herbi.breathe()
```

Output:

```
I eat only plants. I am vegetarian.
I breathe oxygen.
```

Abstract class

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component

Why use Abstract Base Classes:

By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

How Abstract Base classes work:
By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC. ABC works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.

Concrete Methods in Abstract Base Classes:
Concrete classes contain only concrete (normal)methods whereas abstract classes may contain both concrete methods and abstract methods. The concrete class provides an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via super()

Abstract Properties:
Abstract classes include attributes in addition to methods, you can require the attributes in concrete classes by defining them with @abstractproperty.

Abstract method

An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and its abstract methods must be implemented by its subclasses

Interfaces

Interfaces play an important role in software engineering. As an application grows, updates and changes to the code base become more difficult to manage. More often than not, you wind up having classes that look very similar but are unrelated, which can lead to some confusion

Results:

```
#definition of the class starts here
class Person:
    #initializing the variables
    name = ""
    age = 0

    #defining constructor
    def __init__(self, personName, personAge):
        self.name = personName
        self.age = personAge
```

```
    #defining class methods
    def showName(self):
        print(self.name)

    def showAge(self):
        print(self.age)

    #end of the class definition

# Create an object of the class
person1 = Person("Aamir", 20)

#Create another object of the same class
person2 = Person("Ansari", 102)

#call member methods of the objects
person1.showAge()
person2.showName()
```

Output:

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Class-Object-Constructor.py
20
Ansari
```

```
class ClassGrades:

    def __init__(self, grades):
        self.grades = grades

    @classmethod
    def from_csv(cls, grade_csv_str):
        grades = map(int, grade_csv_str.split(', '))
        cls.validate(grades)
        return cls(grades)

    @staticmethod
    def validate(grades):
        for g in grades:
            if g < 0 or g > 100:
```

```
            raise Exception()

try:
    # Try out some valid grades
    class_grades_valid = ClassGrades.from_csv('90, 80, 85, 94, 70')
    print ('Got grades:', class_grades_valid.grades)

    # Should fail with invalid grades
    class_grades_invalid = ClassGrades.from_csv('92, -15, 99, 101, 77, 65, 100')
    print (class_grades_invalid.grades)
except:
    print ('Invalid!')
```

Output:

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Class_Static-method.py
Got grades: <map object at 0x0000022D66F82F10>
Invalid!
```

```
class Country:
    def ShowCountry(self):
        print("This is India");

# inheritance
class State(Country):
    def ShowState(self):
        print("This is State");

st = State();

st.ShowCountry();

st.ShowState();
```

Output:

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Singe-inheritance.py
This is India
This is State
```

```
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))
```

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Multiple_inheritance.py
30
200
0.5
```

```
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()
```

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Multilevel_inheritance.py
dog barking
Animal Speaking
Eating bread…
```

```
class A:
    def fun1(self):
        print('feature_1 of class A')

    def fun2(self):
        print('feature_2 of class A')


class B(A):
    # Modified function that is
    # already exist in class A
    def fun1(self):
        print('Modified feature_1 of class A by class  B')

    def fun3(self):
        print('feature_3 of class B')

# Create instance
obj = B()

# Call the override function
obj.fun1()
```

Output:

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Method-overriding.py
Modified feature_1 of class A by class B
```

```
class Employee:
   def Hello_Emp(self,e_name=None):
      if e_name is not None:
         print("Hello "+e_name)
      else:
         print("Hello ")
emp1=Employee()
emp1.Hello_Emp()
emp1.Hello_Emp("Aamir")
```

Output:

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Method-overloading.py
Hello
Hello Aamir
```

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
   @abstractmethod
   def do_something(self):
      print("Some implementation!")

class AnotherSubclass(AbstractClassExample):
   def do_something(self):
      super().do_something()
      print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()
```

Output:

```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_06\Code>python Abstract.py
Some implementation!
The enrichment from AnotherSubclass
```

Conclusion:

Hence we have successfully understood and used the OOP concepts in python