Experiment 06

Write python programs to understand:
5.1) Classes, Objects, Constructors, Inner class and Static method, class variables
5.2) Different types of Inheritance
5.3) Polymorphism using Operator overloading, Method overloading, Method overriding, Abstract class, Abstract method and Interfaces in Python.

Roll No.	61
Name	V Krishnasubramaniam
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:

Write python programs to understand
5.1) Classes, Objects, Constructors, Inner class and Static method, class variables
5.2) Different types of Inheritance
5.3) Polymorphism using Operator overloading, Method overloading, Method overriding, Abstract class, Abstract method and Interfaces in Python.

Introduction:

Classes and Objects in Python

Class: Classes provide a means of bundling data and functionality together. A class is a collection of related objects having common characteristics and behaviour. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Object: An object is any identifiable entity which has some characteristics and behaviour of its own. An Object is an instance of a Class. An object consists of:

1. State: It is represented by the attributes or the properties of an object.
2. Behaviour: It is represented by the methods of an object.

Creating Classes in Python: The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon.

Syntax:

```
class ClassName:
        'Optional class documentation string'
        class_suite
```

The class has a documentation string, which can be accessed via ClassName.__doc__. The class_suite consists of all the component statements defining class members, data attributes and functions.

Any variable defined in the scope of the class in called a data member of that class. Defining methods in class is done by including the function definitions within the scope of the class block.

Example of a simple class:

```
class MyClass:
   """A simple example class"""
   i = 12345
   def f(self):
      print('hello world')
```

Creating Objects in Python: A class needs to be instantiated if we want to use the class attributes in another class or method. Object of a class can be created by calling the class using the class

name. The class data members and the member methods can be accesses using this object and the dot operator.

Syntax:
<object-name> = <class-name>(<arguments>)
<object-name>.<variableName>

Example:
```
class MyClass:
   """A simple example class"""
   i = 12345
   def f(self):
      return 'hello world'
x = MyClass()
print(x.f())
```

Output:
hello world

Constructors in Python

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:
```
def __init__(self, <parameters>):
        # body of the constructor
```

Types of Constructors:

1. Default constructor: The default constructor is simple constructor which doesn't accept any arguments.It's definition has only one argument which is a reference to the instance being constructed.

   Example:
   ```
   class MyClass:
      x = 0
      #default constructor
      def __init__(self):
         self.x = "Hello"
      def display(self):
         return self.x
   obj = MyClass()
   print(obj.f())
   ```

   Output:

Hello

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:

```
class MyClass:
    x = 0
    #parameterized constructor
    def __init__(self, s):
        self.x = s
    def display(self):
        return self.x
obj = MyClass("Welcome")
print(obj.display())
```

   Output:
   hello world

Inner Class in Python

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:

```
# create a Color class
class Color:
  # constructor
  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()
outer.show()

# create a Lightgreen inner class object
g = outer.lg
g.display()
```

Output:

```
Name: Green
Name: Light Green
Code: 024avc
```

Types of Inner Classes:

1. Multiple inner class: The class contains one or more inner classes is known as multiple inner class. We can have multiple inner class in a class, it is easy to implement multiple inner class.

   Example:

```
# create outer class
class Doctors:
  def __init__(self):
     self.name = 'Doctor'
     self.den = self.Dentist()
     self.car = self.Cardiologist()
  def show(self):
    print('In outer class')
    print('Name:', self.name)
  # create a 1st Inner class
  class Dentist:
      def __init__(self):
         self.name = 'Dr. Savita'
         self.degree = 'BDS'
      def display(self):
         print('In inner class 1')
         print("Name:", self.name)
         print("Degree:", self.degree)
```

```
    # create a 2nd Inner class
    class Cardiologist:
        def __init__(self):
            self.name = 'Dr. Amit'
            self.degree = 'DM'
        def display(self):
            print('In inner class 2')
            print("Name:", self.name)
            print("Degree:", self.degree)

# create a object of outer class
outer = Doctors()
outer.show()
# create a object of 1st inner class
d1 = outer.den
# create a object of 2nd inner class
d2 = outer.car
print()
d1.display()
print()
d2.display()
```

Output:

```
In outer class
Name: Doctor

In inner class 1
Name: Dr. Savita
Degree: BDS

In inner class 2
Name: Dr. Amit
Degree: DM
```

2. Multilevel inner class: The class contains inner class and that inner class again contains another inner class, this hierarchy is known as multilevel inner class.

Example:

```
# create a outer class
class MyClass:
    def __init__(self):
        # create a inner class object
        self.inner = self.Inner()
    def show(self):
```

```
    print('This is an outer class')

  # create a 1st inner class
  class Inner:
    def __init__(self):
      # create a inner class of inner class object
      self.innerclassofinner = self.Innerclassofinner()
    def show(self):
      print('This is the inner class')

    # create a inner class of inner
    class Innerclassofinner:
      def show(self):
        print()
      def show(self):
        print('This is an inner class of inner class')

# create a outer class object
outer = MyClass()
outer.show()
# create a inner class object
obj1 = outer.inner
obj1.show()
# create a inner class of inner class object
obj2 = outer.inner.innerclassofinner
obj2.show()
```

Output:

```
This is an outer class
This is the inner class
This is an inner class of inner class
```

Static Methods in Python

Static methods in Python are extremely similar to python class level methods, the difference being that a static method is bound to a class rather than the objects for that class. This means that a static method can be called without an object for that class. This also means that static methods cannot modify the state of an object as they are not bound to it. Let's see how we can create static methods in Python. Note that in a static method, we don't need the self to be passed as the first argument.

Python Static methods can be created in two ways:

1. Using staticmethod():

The staticmethod() built-in function returns a static method for a given function. But, using staticmethod() is considered an un-Pythonic way of creating a static function. Hence, in newer versions of Python, you can use the @staticmethod decorator.

Example:

```
class Calculator:
    def addNumbers(x, y):
        return x + y
#create addNumbers static method
Calculator.addNumbers = staticmethod(Calculator.addNumbers)
print('Product:', Calculator.addNumbers(15, 110))
```

Output:

```
Product: 125
```

2. Using @staticmethod:

The @staticmethod is a built-in decorator that defines a static method in the class in Python. A static method doesn't receive any reference argument whether it is called by an instance of a class or by the class itself.

Example:

```
class Student:
    name = 'unknown' # class attribute
    def __init__(self):
        self.age = 20  # instance attribute
    @staticmethod
    def tostring():
        print('Student Class')

Student.tostring()
```

Output:

```
Student Class
```

Class Variables in Python

Class variables are declared inside the class definition (but outside any of the instance methods). They are not tied to any particular object of the class, hence shared across all the objects of the class. Modifying a class variable affects all objects instance at the same time. In contrast, instance variables are declared inside the constructor method of class (the __init__ method). They are tied to the particular object instance of the class.

Example:

```
class MyClass:
  x = 10
```

```
  def __init__(self,a):
    self.y = a
obj1 = MyClass(5)
print(obj1.x,obj1.y)
obj2 = MyClass(200)
print(obj2.x,obj2.y)
```

Output:

```
10 5
10 200
```

Unlike Java or C++, Python does not have keywords like public, private or protected. Everything written under class will come under public. If we don't want to make a variable available outside the class or to other members inside the class, we can write two double scores before it as: __var. This is like private variable in python. In order to access such a variable we require Name Mangling.

Example:

```
class MyClass:
def __init__(self):
self.x=1          #public var
self.__y=2        #private var
def display(self):
print(self.x)                    #x is available directly
print(self._MyClass__y)          #name mangling required
print('Accessing variables through method:')
m=MyClass()
m.display()
print('Access variables through instance:')
print(m.x)
print(m._MyClass__y)             #name mangling required
```

Output:

```
Accessing variables through method:
1
2
Access variables through instance:
1
2
```

Inheritance in Python

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

Sub-class is the class that inherits properties from another class is called Sub class or Derived Class. Super Class is the class whose properties are inherited by sub class is called Base Class or Super class.

Advantages of Inheritance:

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.

Example:

```
class Person():
    def __init__(self, name):
        self.name = name
    def getName(self):
        return self.name
    def isTeacher(self):
        return False
class Teacher(Person):
    def isTeacher(self):
        return True
t = Person("Hello1")
print(t.getName(), t.isTeacher ())
t = Employee("Hello2")
print(t.getName(), t.isTeacher ())
```

Output:

```
Hello1 False
Hello2 True
```

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.

   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.
```

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.

Example:

```
class Mother:
   mothername = ""
   def mother(self):
      print(self.mothername)
class Father:
   fathername = ""
   def father(self):
      print(self.fathername)
class Son(Mother, Father):
   def parents(self):
      print("Father :", self.fathername)
      print("Mother :", self.mothername)
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

```
Father : RAM
Mother : SITA
```

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.

Example:

```
class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername
        Grandfather.__init__(self, grandfathername)
class Son(Father):
    def __init__(self,sonname, fathername, grandfathername):
        self.sonname = sonname
        Father.__init__(self, fathername, grandfathername)
    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

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.

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.
```

5. Hybrid Inheritance

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

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()
```

Output:

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

Polymorphism in Python

Polymorphism is the ability of an entity like object, variable or method to take more than one form, by sharing the same external interface while differing in their internal structure. Python has built-in polymorphism. Polymorphism in Python can be implemented in the following ways:

1. **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.

   Example:

```
print(1 + 2)
print("Hello"+"World")
print([0,2,4] + [1,3,5])
print(3 * 4)
print("Hello"*4)
```

   Output:

```
3
HelloWorld
[0,2,4,1,3,5]
12
HelloHelloHelloHello
```

   By default, we cannot use + operator to add two objects like obj1 + obj2. But we can explicitly overload the + symbol.

   Internally, + operator is written as a special __add__() function. Like when we perform a+b, internally a.__add__(b) is being called. We can override this method to act upon our user-defined class objects.

   Syntax:

```
def __add__(self,other):
return self.value + other.value
```

   Example:

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Hello")
ob4 = A("World")
```

```
print(ob1 + ob2)
print(ob3 + ob4)
```

Output:

```
3
HelloWorld
```

Similarly, we can overload other operators like relational operators, by using their respective 'magic' methods:

Operator	Magic Method	Operator	Magic Method
+	__add__(self, other)	<	__lt__(self, other)
-	__sub__(self, other)	>	__gt__(self, other)
*	__mul__(self, other)	<=	__le__(self, other)
/	__div__(self, other)	>=	__ge__(self, other)
//	__floordiv__(self, other)	==	__eq__(self, other)
%	__mod__(self, other)	!=	__ne__(self, other)
**	__pow__(self, other)	~	__invert__(self, other)

Example:

```
class A:
    def __init__(self, a):
        self.a = a
    def __lt__(self, other):
        if(self.a<other.a):
            return "ob1 is lessthan ob2"
        else:
            return "ob2 is less than ob1"
    def __eq__(self, other):
        if(self.a == other.a):
            return "Both are equal"
        else:
            return "Not equal"
ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)
ob3 = A(4)
ob4 = A(4)
print(ob1 == ob2)
```

Output:

```
ob1 is lessthan ob2
Not equal
```

2. **Method Overloading**

When a class has more than one method having the same name, differing only by their parameters, it is called method overloading. Python does not support method overloading by default. Writing more than one method with the same name is not possible in Python. But there are different ways to achieve method overloading in Python.

We can achieve method overloading by writing same method with several parameters. The method performs the operation depending on the number of arguments passed in the method call.

Example:

```
class MyClass:
def sum(self, a=None, b=None, c=None):
if a!=None and b!=None and c!=None:
print(a+b+c)
elif a!=None and b!=None:
print(a+b)
else:
print("Enter atleast 2 arguments")
obj = MyClass()
obj.sum(1,2,3)
obj.sum(1,2)
obj.sum(1)
```

Output:

```
6
3
Enter atleast 2 arguments
```

3. Method Overriding

When a method in a subclass has the same name, same parameters or signature and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class, and this process is called method overriding.

Example:

```
class Parent():
    def __init__(self):
        self.value = "Inside Parent"
    def show(self):
        print(self.value)
class Child(Parent):
    def __init__(self):
        self.value = "Inside Child"
    def show(self):
        print(self.value)
obj1 = Parent()
```

```
obj2 = Child()
obj1.show()
obj2.show()
```

Output:

```
Inside Parent
Inside Child
```

Method overriding with multiple inheritance: When a class is derived from more than one base class it is called multiple Inheritance. Let's consider an example where we want to override a method of one parent class only.

Example:

```
class Parent1():
    def show(self):
        print("Inside Parent1")
class Parent2():
    def display(self):
        print("Inside Parent2")
class Child(Parent1, Parent2):
    def show(self):
        print("Inside Child")
obj = Child()
obj.show()
obj.display()
```

Output:

```
Inside Child
Inside Parent2
```

Method overriding with multilevel inheritance: When we have a child and grandchild relationship. Let's consider an example where we want to override only one method of one of its parent classes.

Example:

```
class Parent():
    def display(self):
        print("Inside Parent")
class Child(Parent):
    def show(self):
        print("Inside Child")
class GrandChild(Child):
    def show(self):
        print("Inside GrandChild")
g = GrandChild()
g.show()
```

```
g.display()
```

Output:

```
Inside GrandChild
Inside Parent
```

Abstract Method and Abstract Class

An abstract method is a method whose action is redefined in the sub-classes as per the requirement of the objects, that is, an abstract method is a method that has a declaration but does not have an implementation. To mark a method as abstract, we should use the decorator @abstractmethod.

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. Once an abstract class is written, we should create sub classes and all the abstract methods should be implemented (body should be written) in the sub-classes. Then it is possible to create the objects to the sub-classes.

Need for Abstract Class: 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.

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.

Example:

```
from abc import ABC, abstractmethod
class Polygon(ABC):
   @abstractmethod
   def noofsides(self):
      pass
class Triangle(Polygon):
   def noofsides(self):
      print("I have 3 sides")
class Quadrilateral(Polygon):
   def noofsides(self):
      print("I have 4 sides")
class Pentagon(Polygon):
   def noofsides(self):
      print("I have 5 sides")
```

```
class Hexagon(Polygon):
   def noofsides(self):
      print("I have 6 sides")
R1 = Triangle()
R1.noofsides()
R2 = Quadrilateral()
R2.noofsides()
R3 = Pentagon()
R3.noofsides()
R4 = Hexagon()
R4.noofsides()
```

Output:

```
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

Interfaces in Python

A class which contains only abstract methods and has no concrete methods becomes an interface, that is, an interface is an abstract class with only abstract methods. Unlike Java, interface concept is not explicitly available in Python, hence abstract classes are used as interfaces. Since an interface contains methods without body, it is not possible to create objects to an interface, we can only be inherited by sub-classes.

It is the discretion of the programmer to decide when to use an abstract class and when to go for an interface.

Example:

```
from abc import *
#an interface to connect to any database
class MyClass(ABC):
        @abstractmethod
        def connect(self):
                pass
        @abstractmethod
        def disconnect(self):
                pass
class MySQL(MyClass):
def connect(self):
print("Connected to MySQL database")
def disconnect(self):
print("Disconnected from MySQL database")
```

```
class Oracle(MyClass):
def connect(self):
print("Connected to Oracle database")
def disconnect(self):
print("Disconnected from Oracle database")
obj1 = MySQL()
obj1.connect()
obj1.disconnect()
obj2 = Oracle()
obj2.connect()
obj2.disconnect()
```

Output:

```
Connected to MySQL database
Disconnected from MySQL database
Connected to Oracle database
Disconnected from Oracle database
```

Results:

Program in class_example_1.py:

```
class MyClass:
  """A simple example class"""
  i = 12345
  def f(self):
    return 'hello world'
x = MyClass()
print(x.f())
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_1.py
hello world
```

Program in class_example_2.py:

```
class MyClass:
    x = 0
    #parameterized constructor
    def __init__(self, s):
        self.x = s
    def display(self):
        return self.x
obj = MyClass("Welcome")
print(obj.display())
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_2.py
Welcome
```

Program in class_example_3.py:

```
# create a Color class
class Color:
  # constructor
  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()
outer.show()

# create a Lightgreen inner class object
g = outer.lg
g.display()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_3.py
Name: Green
Name: Light Green
Code: 024avc
```

Program in class_example_4.py:

```
# create outer class
class Doctors:
  def __init__(self):
```

```
    self.name = 'Doctor'
    self.den = self.Dentist()
    self.car = self.Cardiologist()

  def show(self):
    print('In outer class')
    print('Name:', self.name)

  # create a 1st Inner class
  class Dentist:
      def __init__(self):
          self.name = 'Dr. Savita'
          self.degree = 'BDS'
      def display(self):
          print('In inner class 1')
          print("Name:", self.name)
          print("Degree:", self.degree)

  # create a 2nd Inner class
  class Cardiologist:
      def __init__(self):
          self.name = 'Dr. Amit'
          self.degree = 'DM'
      def display(self):
          print('In inner class 2')
          print("Name:", self.name)
          print("Degree:", self.degree)

# create a object of outer class
outer = Doctors()
outer.show()

# create a object of 1st inner class
d1 = outer.den

# create a object of 2nd inner class
d2 = outer.car
print()
d1.display()
print()
d2.display()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_4.py
In outer class
Name: Doctor

In inner class 1
Name: Dr. Savita
Degree: BDS

In inner class 2
Name: Dr. Amit
Degree: DM
```

Program in class_example_5.py:

```
# create a outer class
class MyClass:
  def __init__(self):
     # create a inner class object
     self.inner = self.Inner()
  def show(self):
     print('This is an outer class')

  # create a 1st inner class
  class Inner:
    def __init__(self):
      # create a inner class of inner class object
      self.innerclassofinner = self.Innerclassofinner()
    def show(self):
      print('This is the inner class')

    # create a inner class of inner
    class Innerclassofinner:
      def show(self):
        print()
      def show(self):
        print('This is an inner class of inner class')

# create a outer class object
outer = MyClass()
outer.show()
# create a inner class object
obj1 = outer.inner
obj1.show()
# create a inner class of inner class object
obj2 = outer.inner.innerclassofinner
obj2.show()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_5.py
This is an outer class
This is the inner class
This is an inner class of inner class
```

Program in class_example_6.py:

```
class Calculator:
    def addNumbers(x, y):
        return x + y

# create addNumbers static method
Calculator.addNumbers = staticmethod(Calculator.addNumbers)

print('Product:', Calculator.addNumbers(15, 110))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_6.py
Product: 125
```

Program in class_example_7.py:

```
class Student:
    name = 'unknown' # class attribute
    def __init__(self):
        self.age = 20  # instance attribute
    @staticmethod
    def tostring():
        print('Student Class')
Student.tostring()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_7.py
Student Class
```

Program in class_example_8.py:

```
class MyClass:
  x = 10
  def __init__(self,a):
    self.y = a
obj1 = MyClass(5)
print(obj1.x,obj1.y)
obj2 = MyClass(200)
print(obj2.x,obj2.y)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_8.py
10 5
10 200
```

Program in class_example_9.py:

```
class MyClass:
def __init__(self):
self.x=1 #public var
self.__y=2 #private var
def display(self):
print(self.x) #x is available directly
print(self._MyClass__y)        #name mangling required
print('Accessing variables through method:')
m=MyClass()
m.display()
print('Access variables through instance:')
print(m.x)
print(m._MyClass__y) #name mangling required
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python class_example_9.py
Accessing variables through method:
1
2
Access variables through instance:
1
2
```

Program in inherit1.py:

```
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:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python inherit1.py
This function is in parent class.
This function is in child class.
```

Program in inherit2.py:

```
class Mother:
   mothername = ""
```

```
    def mother(self):
        print(self.mothername)
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python inherit2.py
Father : RAM
Mother : SITA
```

Program in inherit3.py:

```
class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername
        Grandfather.__init__(self, grandfathername)
class Son(Father):
    def __init__(self,sonname, fathername, grandfathername):
        self.sonname = sonname
        Father.__init__(self, fathername, grandfathername)
    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python inherit3.py
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

Program in inherit4.py:

```
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:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python inherit4.py
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Program in inherit5.py:

```
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()

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>python inherit5.py
This function is in school.
This function is in student 1.
```

Program in op1.py:

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Hello")
ob4 = A("World")
print(ob1 + ob2)
print(ob3 + ob4)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py op1.py
3
HelloWorld
```

Program in op2.py:

```
class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __add__(self, other):
        return self.a + other.a, self.b + other.b
    def __str__(self):
        return self.a, self.b
obj1 = complex(1, 2)
obj2 = complex(2, 3)
obj3 = obj1 + obj2
print(obj3)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py op2.py
(3, 5)
```

Program in op3.py:

```
class A:
    def __init__(self, a):
        self.a = a
    def __lt__(self, other):
        if(self.a<other.a):
            return "ob1 is lessthan ob2"
        else:
            return "ob2 is less than ob1"
    def __eq__(self, other):
        if(self.a == other.a):
            return "Both are equal"
        else:
            return "Not equal"
ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)
ob3 = A(4)
ob4 = A(4)
print(ob1 == ob2)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py op3.py
ob1 is lessthan ob2
Not equal
```

Program in op4.py:

```
class Employee:
    def __init__(self,name,salary):
        self.name=name
        self.salary = salary
    def __mul__(self,other):
        return self.salary * other.days
class Attendance:
    def __init__(self, name, days):
        self.name = name
        self.days = days
obj1 = Employee('Srinu',500.0)
obj2 = Attendance('Srinu',25)
print("Salary =",obj1*obj2)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py op4.py
Salary = 12500.0
```

Program in methodoverload1.py:

```
class MyClass:
def sum(self, a=None, b=None, c=None):
if a!=None and b!=None and c!=None:
print(a+b+c)
elif a!=None and b!=None:
print(a+b)
else:
                print("Enter atleast 2 arguments")
obj = MyClass()
obj.sum(1,2,3)
obj.sum(1,2)
obj.sum(1)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py methodoverload1.py
6
3
Enter atleast 2 arguments
```

Program in methodoverride1.py:

```
class Parent():
    def __init__(self):
        self.value = "Inside Parent"
    def show(self):
        print(self.value)
class Child(Parent):
    def __init__(self):
        self.value = "Inside Child"
    def show(self):
        print(self.value)
obj1 = Parent()
obj2 = Child()
obj1.show()
obj2.show()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py methodoverride1.py
Inside Parent
Inside Child
```

Program in methodoverride2.py:

```
class Parent1():
    def show(self):
        print("Inside Parent1")
class Parent2():
    def display(self):
        print("Inside Parent2")
class Child(Parent1, Parent2):
    def show(self):
        print("Inside Child")
obj = Child()
obj.show()
obj.display()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py methodoverride2.py
Inside Child
Inside Parent2
```

Program in methodoverride3.py:

```
class Parent():
    def display(self):
        print("Inside Parent")
class Child(Parent):
    def show(self):
        print("Inside Child")
class GrandChild(Child):
    def show(self):
        print("Inside GrandChild")
g = GrandChild()
g.show()
g.display()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py methodoverride3.py
Inside GrandChild
Inside Parent
```

Program in abstract1.py:

```
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def noofsides(self):
        pass
```

```
class Triangle(Polygon):
    def noofsides(self):
        print("I have 3 sides")

class Quadrilateral(Polygon):
    def noofsides(self):
        print("I have 4 sides")

class Pentagon(Polygon):
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(Polygon):
    def noofsides(self):
        print("I have 6 sides")

R1 = Triangle()
R1.noofsides()
R2 = Quadrilateral()
R2.noofsides()
R3 = Pentagon()
R3.noofsides()
R4 = Hexagon()
R4.noofsides()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py abstract1.py
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

Program in abstract2.py:

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(Self):
        pass

class Bus(Vehicle):
    def getNoOfWheels(self):
        return 6
```

```
class Auto(Vehicle):
   def getNoOfWheels(self):
      return 3

b=Bus()
print(b.getNoOfWheels())
a=Auto()
print(a.getNoOfWheels())
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py abstract2.py
6
3
```

Program in interface1.py:

```
from abc import *
#an interface to connect to any database
class MyClass(ABC):
@abstractmethod
def connect(self):
pass
@abstractmethod
def disconnect(self):
pass
class MySQL(MyClass):
def connect(self):
print("Connected to MySQL database")
def disconnect(self):
print("Disconnected from MySQL database")
class Oracle(MyClass):
def connect(self):
print("Connected to Oracle database")
def disconnect(self):
print("Disconnected from Oracle database")
obj1 = MySQL()
obj1.connect()
obj1.disconnect()
obj2 = Oracle()
obj2.connect()
obj2.disconnect()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py interface1.py
Connected to MySQL database
Disconnected from MySQL database
Connected to Oracle database
Disconnected from Oracle database
```

Program in interface2.py:

```
from abc import ABC, abstractmethod
class Bank(ABC):
  @abstractmethod
  def balance_check(self):
    pass
  @abstractmethod
  def interest(self):
    pass
class SBI(Bank):
  def balance_check(self):
    print("Balance is 100 rupees")
  def interest(self):
    print("SBI interest is 5 rupees")
s = SBI()
s.balance_check()
s.interest()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 6>py interface2.py
Balance is 100 rupees
SBI interest is 5 rupees
```

Conclusion:

Thus, we have understood the basic Object-Oriented Concepts of objects, classes, inheritance and polymorphism. We also learnt and performed hands-on practical program about inner classes, abstract classes, interfaces, types of methods, types of variables, types of inheritance and various ways to implement the concept of polymorphism like method overloading, operator overloading and method overriding.