UNIT 2

BASICS OF OBJECT ORIENTED PROGRAMMING

2.1 Oops Concepts :

2.1.1 Data Abstraction :

- Data abstraction refers to the process of providing only essential information to the outside world and hiding their background details. Thus only that information is represented that is needed without explaining the details.
- For Example consider a TV in which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players BUT you do not know it's internal detail that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.
- Thus we can say, a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.
- Now if we talk in terms of Python Programming, Python classes provides great level of **data abstraction.** They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data i.e. State without actually knowing how class has been implemented internally.

2.1.2 Data Encapsulation :

- The process of binding Attributes (Variables) and its associated Method (function) into a single unit is known as data encapsulation.
- Thus Attributes(Variables) can be accessed only by the Method(function) that is bind with it. It cannot be accessed outside the Class.
- Thus Data Encapsulation provides data hiding facility.

2.1.3 Class & Object :

- A class is a user defined data structure that allows us to bind attributes (Variable) and its associat methods(functions) together as a single unit. Thus class provides the facility of data encapsulatio
- Once a class is defined we can create an object of the class to access attributes(variables) a methods(functions) defined inside the class. Thus we can say that an object is an instance of t class.

2.1.4 Constructor :

- Constructor is a method of the class. It is called special method of the class because its nam is **__init__()**.
- Constructor is used to construct the values for the attributes (Variables) of the class automatica when the object of class is created.
- Like other method of the class there is no need to call constructor explicitly. It is invoked automatica each time the object of its class is created.
- In python basically there are three types of constructors:
  (1) Default constructor
  (2) Non Parameterized Constructor
  (3) Parameterized Constructor

2.1.5 Inheritance :

- The process of deriving a new class called the derived class from already existing class is know as inheritance.
- It provides the concept of reusability. I.e. we can add new features to the existing class witho modifying it.
- There are various types of inheritance:
  (1) Single (2) Multiple (3) Multilevel (4) Hybrid (5) Hierarchical

2.1.6 Polymorphism :

- The process of using same thing for different purpose is known as polymorphism.
- It simply means one name multiple forms.

2.2 Class and Object :

- A class is a user defined data structure that allows us to bind attributes (Variable) and its associate methods(functions) together as a single unit. Thus class provides the facility of data encapsulation.
- Attributes (Variables) of the class are public by default.
- The general syntax for declaring a class is given below:

```
class Class_Name:
    Attribute Declaration
    Method Definition
```

- Once a class is defined we can create an object of the class to access attributes(variables) and methods(functions) defined inside the class. Thus we can say that an object is an instance of the class.
- General Syntax for creating an object is given below:

```
Object_Name = Class_Name([Argument_List])
```

- Consider following example :

```
class  Student:
        RollNumber=0
        Name=""
        def   DisplayStudent(self):
              print("RollNumber={}".format(self.RollNumber))
              print("Name={}".format(self.Name))
S1  =  Student()
S1.RollNumber  =  int(input("Enter  RollNumber"))
S1.Name  =  input("Enter  Name")
S1.DisplayStudent()
```

- In above example we define a class named Student using class keyword. Student class consist of two attributes (RollNumber and Name) and one method(DisplayStudent).
- The **self** parameter defined in DisplayStudent() method refers to the current instance of the class. It contains reference of the object using which method is called. Using **self** parameter we can access attributes(Variables) of the class. Every methods of the class must have **self** parameter as a first parameter in method definition. There is no need to pass **self** parameter while calling the method. In order to access class attributes inside method we must have to define self parameter as a first parameter of the method.
- After defining a class we create an object S1 of class Student using following statement:

```
S1  =  Student()
```

- After creating an object S1 we access attributes and methods of class Student.
- By default each member of the class are public.

2.3 Constructors :

IMP

- Constructor is a method of the class. It is called special method of the class because its name is __init__().
- Constructor is used to construct the values for the attributes (Variables) of the class automatically when the object of class is created.
- Like other method of the class there is no need to call constructor explicitly. It is invoked automatically each time the object of its class is created.
- Every class having at least one constructor defined in it. If you do not define any constructor in the class then interpreter will automatically create a constructor inside the class and assigns default value to the attributes (Variables) of the class.
- Constructor can be defined inside class as shown below :

```
class  Class_Name:
      def   __init__(self):
            self.Attribute_Name1  =  Value1
            self.Attribute_Name2  =  Value2
```

- Consider Following Example :

```
class  Student:
      def   __init__(self):
            self.RollNumber=1
```

```
            self.Name="Yesha"
        def DisplayStudent(self):
            print("RollNumber={}".format(self.RollNumber))
            print("Name={}".format(self.Name))
S1 = Student()
S1.DisplayStudent()
```

- In above example Class Student having constructor named __init__() which is used to initializ attributes (Variables) of the class. It is invoked automatically when object of the class Stude is created.
- In above example __init__() method has only one parameter named self which refers to the obje (S1) using which this method is called. It is also known as non parameterized constructor.
- In python basically there are three types of constructors :
  (1) Default constructor
  (2) Non Parameterized Constructor
  (3) Parameterized Constructor

(1) Default Constructor :

- Every class having at least one constructor defined in it. If you do not define any constructo in the class then interpreter will automatically create a constructor inside the class and assigns default value to the attributes (Variables) of the class.
- Consider Following Example:

```
class Student:
    RollNumber=1
    Name="Yesha"
    def DisplayStudent(self):
        print("RollNumber={}".format(self.RollNumber))
        print("Name={}".format(self.Name))
S1 = Student()
S1.DisplayStudent()
```

- In above example class Student having two attributes named RollNumber and Name which are assigned default value 1 and "Yesha" respectively at the time of defining class. As you can see there is no __init__() method defined inside class even though each time we create an instance of class student it will automatically assigns default value to the attributes of class for that instance. It is possible because python creates **default constructor** at run time and assigns default value to each attributes of the class.

(2) Non Parameterized Constructor :

- A Constructor **__init__()** that has only one parameter named **self** is called Non Parameterized Constructor. There is no need to pass any argument while creating an object using this constructor.
- Consider Following Example:

```
class Student:
    def __init__(self):
        self.RollNumber=1
        self.Name="Yesha"
```

```
    def DisplayStudent(self):
        print("RollNumber={}".format(self.RollNumber))
        print("Name={}".format(self.Name))
S1 = Student()
S1.DisplayStudent()
```

- In above example class Student having constructor named **__init__()** which has only one parameter named **self**. Each time an object of class student is created this constructor is invoked automatically and assigns default value to the attributes (Variables) of the class for that object.
- Disadvantage of Non Parameterized Constructor is that each time an object is created it will assign same default values to the attributes (Variables) of the class. It is not possible to assign different values to the attributes (Variables) for the different object of the class using Non Parameterized constructor.

(3) Parameterized Constructor :

- Disadvantage of Non Parameterized Constructor is that each time an object is created it will assign same default values to the attributes (variables) of the class.
- However sometimes it is required to assign different values to the attributes (variables) for the different object of the class. This can be accomplished using the concept of Parameterized constructor.
- The constructor **__init__()** that has parameters other than **self** is called Parameterized constructor. We have to pass arguments at the time of creating an object using this constructor.
- Consider following example :

```
class Student:
    def __init__(self,RNumber,SName):
        self.RollNumber=RNumber
        self.Name=SName
    def DisplayStudent(self):
        print("RollNumber={}".format(self.RollNumber))
        print("Name={}".format(self.Name))
S1 = Student(1,"Yesha")
S1.DisplayStudent()
S2 = Student(2,"Shanvi")
S2.DisplayStudent()
```

- In above example class Student having constructor named **__init__()** which has three parameter named **self, RNumber and SName**. Each time an object of class student is created we have to pass value for **RNumber** and **SName.**

2.4 Types of methods :

- Class is a collection of attributes (Variables) and Methods. It provides the facility of binding attributes and its associated methods together as a single unit. Using Methods you can describe behaviour of class object.
- In Python basically there are three types of Methods available :
  (1) Instance Method
  (2) Class Method
  (3) Static Method

2.4.1 Instance method :

- Instance method is used to get or set attributes of an instance(object). Using Instance method can access various attributes (variables) of an object and can also performs various operations attributes of an object.
- By default all the methods defined inside class is known as Instance Method unless and un is specified.
- Instance Method has default parameter named **self**. self must be the first parameter of Insta method at the time of defining it. self refers to the instance of a class using which the met is called. It is not required to pass this parameter at the time of calling method.
- Instance Method can be invoked using an instance of the class.
- Consider Following Example:

```
class  Student:
        def    __init__(self,Name,Age):
               self.Name=Name
               self.Age=Age
        def    Display(self):
               print("Name:{}".format(self.Name))
               print("Age:{}".format(self.Age))
S  =  Student("Shanvi",8)
S.Display()
```

- In above example class **Student** having one Instance Method named **Display()**. We can acce attributes (Name and Age) of the class using this method. Display() method can be invoked creating an instance of class **Student**.

2.4.2 Class method :

- Class Method is used to get or set details of the class. Class Method cannot access instance specif data. They are limited to deal with class instead of objects.
- Class Method can be defined by specifying **@classmethod** decorator just before definition method.
- Class Method has default parameter named **cls**. cls must be the first parameter of Class Metho at the time of defining it. cls refers to the class in which it is defined. It is not required to pa this parameter at the time of calling method.
- Class Method can be invoked using either instance of the class or using Class itself.
- Consider Following Example :

```
class  Student:
       SchoolName="My  School"
       @classmethod
       def    ChangeSchoolName(cls,SchoolName):
              cls.SchoolName=SchoolName
       @classmethod
       def    DisplaySchool(cls):
              print(cls.SchoolName)
Student.DisplaySchool()
Student.ChangeSchoolName("Your  School")
Student.DisplaySchool()
```

- In above example class **Student** having two Class Methods named **DisplaySchool()** and **ChangeSchoolName()**. We can access class specific attributes such as **SchoolName** using this methods. Class Methods can be directly invoked using class name. If we wish we can invoke Class Method also using an instance of the Class.

2.4.3 Static method :

- Static Method does not access class specific attributes or object specific attributes. They are independent of class state and object state. Static Methods are just like utility methods.
- Static Method can be defined by specifying **@staticmethod** decorator just before definition of method.
- Like Instance Method and Class Method it does not has any default parameter. It can be invoked using Class Name or using an instance of the class.
- Consider Following Example :

```
from date time import date
class  Student:
        def    __init__(self,Name,Age):
               self.Name=Name
               self.Age=Age
        def    Display(self):
               print(self.Name)
               print(self.Age)
        @staticmethod
        def    PrintDate():
               today  =  date.today()
               print("Today's  date:",  today)
s  =  Student("Yesha",15)
s.Display()
Student.PrintDate()
```

- In above example class **Student** having one instance method named **Display()** and one Static Method named **PrintDate()**.

2.5 Data Encapsulation :

- The process of binding attributes (Variables) and its associated Methods (Functions) into a single unit called Class is known as data encapsulation.
- Thus the attributes (Variables) can be accessed only by the Methods (Functions) that is bind with it. It cannot be accessed outside the Class.
- Thus Data Encapsulation provides data hiding facility. Data encapsulation can be achieved using access modifier in python.

2.5.1 Access Modifier in Class :

- Access Modifiers are used to restrict access of members of the class. Using the concept of Access Modifier Object Oriented Languages provides the facility of Data Hiding. Access Modifier defines how members of the class can be accessed.
- Python supports three types of access modifier :

(1) Public (Default for each Member of Class)

- Public Data Member of the class can be accessed from outside class using object of the class.

- By Default all the members of the class are Public.
- Public data member of the class can be inherited by derived class.

(2) Protected

- Protected Data Member of the class can be accessed from outside class but by only th class which is derived from the class in which it is declared as Protected.
- Data Member can be declared as Protected by proceeding it with **Single Underscore (Symbol.**
- Consider Following Example:

```
class Base:
    def __init__(self):
        self._BaseAttribute="Base"
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self._DerivedAttribute="Derived"
D = Derived()
print(D._BaseAttribute)
print(D._DerivedAttribute)
D._BaseAttribute="Base Modified"
print(D._BaseAttribute)
```

- In above example class **Base** having one protected attribute named _BaseAttribute. It ca be accessed from Derived class named **Derived** and also using an object **D** of class **Derive**

(3) Private

- Private Data Member of the class can only be accessed from within class. It cannot b accessed from outside class.
- Data Member can be declared as Private by proceeding it with **Double Underscore (Symbol.**
- Private Data Member of the class can be accessed by only public methods of the clas It cannot be accessed using object of the class.
- Private Data member of the class cannot be inherited by derived class.
- Consider Following Example :

```
class Base:
    def __init__(self):
        self.__BaseAttribute="Base"
B = Base()
print(B.__BaseAttribute)
```

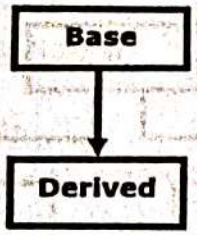
- Here, Base class having one private attribute named __BaseAttribute. Trying to access using object of the Base class B will raise an error.
- In order to access private attribute of class we need to define public method which ca access it using object of the class. Consider following example:

```
class Base:
```

```
        def __init__(self):
            self.__BaseAttribute="Base"
        def Display(self):
            print(self.__BaseAttribute)
B = Base()
B.Display()
```

2.6 Inheritance - single, multiple, multi-level, hierarchical, hybrid :

- **Inheritance :** The process of deriving a new class from an already existing class is known as inheritance.
- The class that is derived is known as derived class and the class from which new class is derived is known as base class.



- Most important feature of the inheritance is reusability. It means if a class is designed, compiled and tested we can utilize that class in future. We can also add extra features as per our requirements.
- **How to define derived class**
- First define base class as shown below :

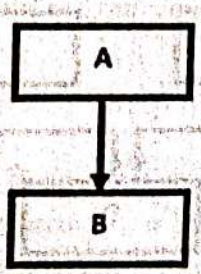
```
class Base_Class_Name:
    Attribute Declaration
    Method Definition
```

- Once base class is defined you can derive new class from base class. General Syntax for deriving new class from base class is given below:

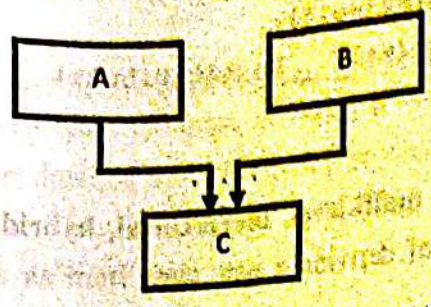
```
class Derived_Class_Name (Base_Class_Name[,Base_Class_ Name]):
    Attribute Declaration
    Method Definition
```

- You can specify more than one base class in case of Multiple Inheritance.
- **Types of Inheritance :**

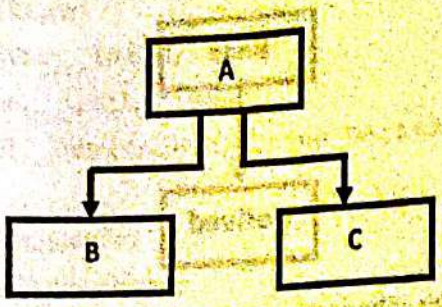
(1) **Single Inheritance :** The process of deriving a new class from already existing class is known as single inheritance. Thus in single inheritance there is one base class and one derived class.



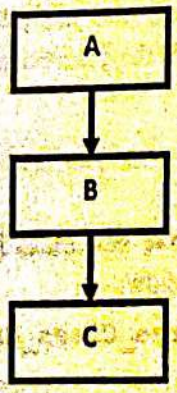
(2) Multiple Inheritances : The process of deriving a new class from more than one base cl is known as multiple inheritance. Thus in multiple inheritance there is one derived class more than one base classes.



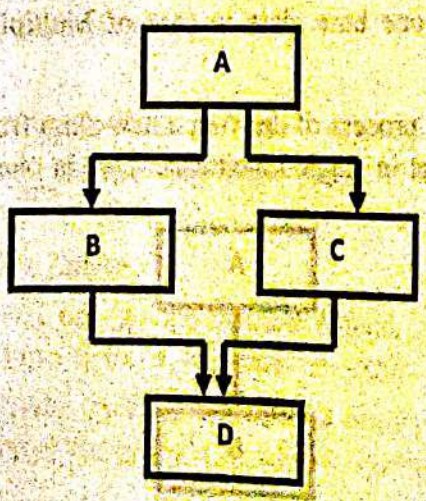
(3) Hierarchical Inheritance : The process of deriving more than one class from single base cl is known as Hierarchical Inheritance. Thus in Hierarchical Inheritance there is one base cl but more than one derived classes.



(4) Multilevel Inheritance : The process of deriving a new class from an already existing cla and then again derive a new class from previously derived class is known as multilevel inheritan

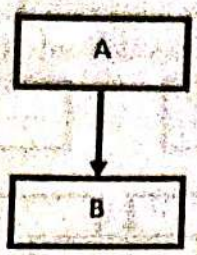


(5) Hybrid Inheritance : The combination of more than one inheritance is known as hybr inheritance.



2.6.1 Single Inheritance :

- The process of deriving a new class from already existing class is known as single inheritance. Thus in single inheritance there is one base class and one derived class.



- Consider following example :

```
class Student:
    def __init__(self,RollNumber,Name):
        self.RollNumber=RollNumber
        self.Name=Name
    def DisplayStudent(self):
        print("Roll Number:{}".format(self.RollNumber))
        print("Name:{}".format(self.Name))
class Result(Student):
    def __init__(self,RollNumber,Name,Marks):
        Student.__init__(self,RollNumber,Name)
        self.Marks=Marks
    def Display(self):
        Student.DisplayStudent(self)
        print("Total Marks:{}".format(self.Marks))
RollNumber = int(input("Enter Roll Number:"))
Name = input("Enter Name:")
TotalMark = int(input("Enter Total Marks:"))
R = Result(RollNumber,Name,TotalMark)
R.Display()
```

- In above example we define a base class named **Student.** It has two public attributes (Roll Number and Name) and one method DisplayStudent(). Next we derive a new class named **Result** from base class **Student.** Result class has one public attribute (Marks) and one method Display(). As Result class is derived from Student class, public attributes(RollNumber and Name) and DisplayStudent() method of Student class also becomes member of Result class. You can access public attributes and methods of base class from derived class using object of derived class or from derived class's methods.
- It is also possible to call base class's constructor method __init__() from derived class's constructor using following syntax :

```
Base_Class_Name.__init__(self[,Arguments])
```

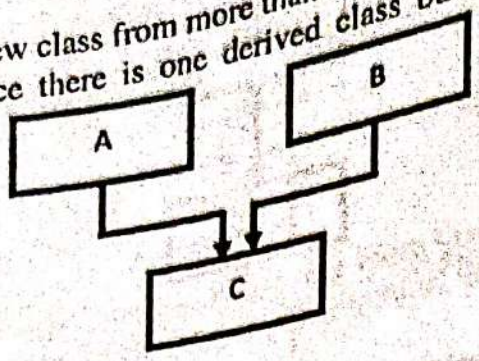
- In above example we call Student class's constructor from Result class using following statement :

```
Student.__init__(self,RollNumber,Name)
```

- In driver code we create an instance of Result class and acces
  Result and Student class.

2.6.2 Multiple Inheritances :

- The process of deriving a new class from more than one base class is known as multiple inher
  Thus in multiple inheritance there is one derived class but more than one base classes.



- Consider following example :

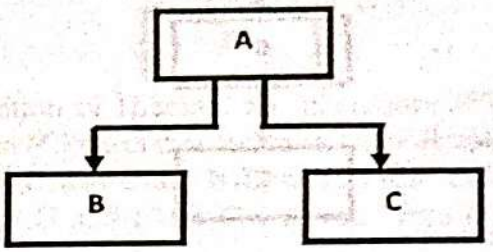
```
class Student:
    def __init__(self,RollNumber,Name):
        self.RollNumber=RollNumber
        self.Name=Name
    def DisplayStudent(self):
        print("Roll Number:{}".format(self.RollNumber))
        print("Name:{}".format(self.Name))
class Exam:
    def __init__(self,Mark1,Mark2,Mark3):
        self.Mark1=Mark1
        self.Mark2=Mark2
        self.Mark3=Mark3
    def DisplayMark(self):
        print("Mark1:{}".format(self.Mark1))
        print("Mark2:{}".format(self.Mark2))
        print("Mark3:{}".format(self.Mark3))
class Result(Student,Exam):
    def __init__(self,RollNumber,Name,Mark1,Mark2,Mark3):
        Student.__init__(self,RollNumber,Name)
        Exam.__init__(self,Mark1,Mark2,Mark3)
    def DisplayResult(self):
        self.Total = self.Mark1+self.Mark2+self.Mark3
        self.Percentage = (self.Total*100)/210
        Student.DisplayStudent(self)
        Exam.DisplayMark(self)
        print("Total Mark:{}".format(self.Total))
        print("Percentage:{}".format(self.Percentage))
RollNumber = int(input("Enter Roll Number:"))
Name = input("Enter Name:")
Mark1=int(input("Enter Mark1:"))
```

```
Mark2=int(input("Enter  Mark2:"))
Mark3=int(input("Enter  Mark3:"))
R  =  Result(RollNumber,Name,Mark1,Mark2,Mark3)
R.DisplayResult()
```

- In above example we define two classes named **Student** and **Exam** and then derive new class named **Result** from Student and Exam class. Here we derived Result class from two base classes so it is known as multiple inheritance. **Result** class can access all the public and protected attributes and methods of Student and Exam class.
- In driver code we create an instance of Result class and access attributes and Methods of all three classes.

2.6.3 Hierarchical Inheritance :

- The process of deriving more than one class from single base class is known as Hierarchical Inheritance. Thus in Hierarchical Inheritance there is one base class but more than one derived classes.



- Consider following example :

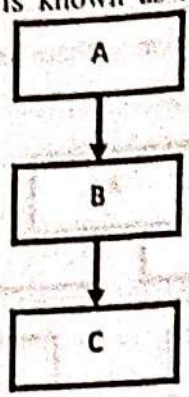
```
class  Company:
      def   __init__(self,CompanyName):
            self.CompanyName=CompanyName
      def   DisplayCompany(self):
            print("Company  Name:{}".format(self.CompanyName))
class  Purchase(Company):
      def   __init__(self,CompanyName,ManagerName):
            Company.__init__(self,CompanyName)
            self.ManagerName=ManagerName
      def   DisplayPurchase(self):
            Company.DisplayCompany(self)
            print("Manager  Name:{}".format(self.ManagerName))
class  Sales(Company):
      def   __init__(self,CompanyName,ManagerName):
            Company.__init__(self,CompanyName)
            self.ManagerName=ManagerName
      def   DisplaySales(self):
            Company.DisplayCompany(self)
            print("Manager  Name:{}".format(self.ManagerName))
CompanyName  =  input("Enter  Company  Name:")
```

```
PManagerName = input("Enter Purchase Manager Name:")
SManagerName = input("Enter Sales Manager Name:")
P = Purchase(CompanyName,PManagerName)
P.DisplayPurchase()
S = Sales(CompanyName,SManagerName)
S.DisplaySales()
```

❖ In above example we define a base class named Company and then derive two classes named Purchase and Sales from it. Purchase class can access public and protected members of Company class. Same way Sales class can also access public and protected members of Company class.

2.6.4 Multilevel Inheritance :

❖ The process of deriving a new class from an already existing class and then again derive a new class from previously derived class is known as multilevel inheritance.



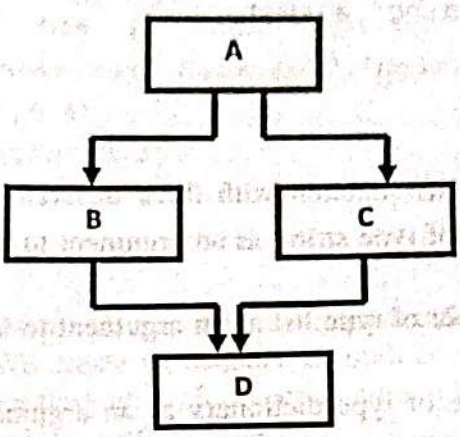
❖ Consider following example :

```
class GrandFather:
    def __init__(self,GrandFatherName):
        self.GrandFatherName=GrandFatherName
class Father(GrandFather):
    def __init__(self,GrandFatherName,FatherName):
        GrandFather.__init__(self,GrandFatherName)
        self.FatherName=FatherName
class Son(Father):
    def __init__(self,GrandFatherName,FatherName,SonName):
        Father.__init__(self,GrandFatherName,FatherName)
        self.SonName=SonName
    def DisplayFullName(self):
        print("My Name Is:{}".format(self.SonName))
        print("My Father Name Is:{}".format(self.FatherName))
        print("My GrandFather Name Is:{}".format(self.GrandFatherName))
Name = input("Enter Your Name:")
FatherName = input("Enter Father Name:")
GrandFatherName = input("Enter GrandFather Name:")
S = Son(GrandFatherName,FatherName,Name)
S.DisplayFullName()
```

- In above example we define a class named **GrandFather** and derive class named **Father** from it. Father class can access all public and protected members of GrandFather class. After deriving Father class we derive class named **Son** from Father class. Here, Son class can access all public members of GrandFather and Father Class. Protected Members of class Father can be accessed by Son class but protected member of GrandFather class cannot be accessed by Son class.

2.6.5 Hybrid Inheritance :

- The combination of more than one inheritance is known as hybrid inheritance.



- Above figure shows combination of Hierarchical Inheritance (Class B and Class C derived from Class A), Multiple Inheritance (Class D derived from Class B and Class C), Multilevel Inheritance (Class D Derived from Class B and Class B Derived From Class A) and Multilevel Inheritance (Class D Derived from Class C and Class C Derived From Class A)

2.7 Polymorphism through inheritance :

- The process of using same thing for different purpose is known as polymorphism.
- It simply means one name multiple forms.

2.7.1 Polymorphism in Addition Operator (+) :

- Basically, addition operator (+) is used for addition of two numbers. But in python depending upon data types it performs different operations.
- When addition operator is used with numeric data type it performs addition of two numbers. Consider following example :

```
a = 5
b = 6
c = a + b
print(str(c))
```

- When addition operator is used with string data type it performs concatenation of two strings. Consider following example :

```
a = "Data Structure"
b = " With Python"
c = a + b
print(c)
```

- So, we can say that addition operator is single but it is used for multiple use depending upon the data type. This is one of the example of Polymorphism in Python.

2.7.2 Polymorphism in Function :

- In Python it is also possible that same function can be used for different task. For Example is a built in function in python. It performs different task depending upon type of value pa to that function. Consider following example:

```
a = "Data Structure"
b = ["A","B","C"]
c = {1:"A",2:"B",3:"C",4:"D"}
print(len(a))
print(len(b))
print(len(c))
```

- In above example we use **len()** function with three different data types.
- In first case we pass variable of type string as an argument to len() function and it returns nun of characters in the string.
- In second case we pass variable of type list as an argument to len() function and it returns nun of elements in the list.
- In third case we pass variable of type dictionary as an argument to len() function and it ret number of keys in the dictionary.
- Thus, we can say that same function is used for different task depending upon the type of v passed to the function. This is an example of function polymorphism. Same way we can also de user defined functions which can be used for different task depending upon the situation in wh it is used.
- Consider Following Example :

```
def Area(a,b=0):
    if(b>0):
        return a*b
    else:
        return a**2
print("Area of Square is {}".format(str(Area(2))))
print("Area of Rectangle is {}".format(str(Area(2,3))))
```

2.7.3 Polymorphism in Class Method :

- In Python it is possible that more than one classes having same methods inside them but hav different implementation. We can group objects of such classes into either list or tuple and th call same methods of different objects by iterating through list or tuple.
- Consider following example :

```
class Square:
    def __init__(self,Name,Side,Angle):
        self.Name = Name
        self.Side=Side
        self.Angle=Angle
    def Display(self):
        print("{} has {} Sides and {}
Angles".format(self.Name,self.Side,self.Angle))
```

```
class Triangle:
    def __init__(self,Name,Side,Angle):
        self.Name = Name
        self.Side=Side
        self.Angle=Angle
    def Display(self):
        print("{} has {} Sides and {}
Angles".format(self.Name,self.Side,self.Angle))
S = Square("Square",4,4)
T = Triangle("Triangle",3,3)
ShapeList = [S,T]
for i in ShapeList:
    i.Display()
```

- In above example we create two classes named **Square** and **Triangle**. Both classes having same method named **Display()**. We create an instance of both classes and then grouped them in to list named **ShapeList**. We iterate through each object using single variable and call **Display()** method. At run time Python will check type of the object and call appropriate version of Display() method.

2.7.4 Polymorphism Using Function with object as argument :

- In Python we can define a function which accepts an object as argument and call method of that object without checking its class type. Thus we can use same function to invoke method of different class by passing object as an argument.
- Consider following example :

```
class Square:
    def __init__(self,Name,Side,Angle):
        self.Name = Name
        self.Side=Side
        self.Angle=Angle
    def Display(self):
        print("{} has {} Sides and {}
Angles".format(self.Name,self.Side,self.Angle))
class Triangle:
    def __init__(self,Name,Side,Angle):
        self.Name = Name
        self.Side=Side
        self.Angle=Angle
    def Display(self):
        print("{} has {} Sides and {}
Angles".format(self.Name,self.Side,self.Angle))
    def ShapeDetail(Obj):
        Obj.Display()
S = Square("Square",4,4)
T = Triangle("Triangle",3,3)
ShapeDetail(S)
ShapeDetail(T)
```

- In above example, we define two classes named **Square** and **Triangle**. Both classes having method named **Display()**. We create a common function named **ShapeDetail()** which accepts as argument and call method associated with that object.

2.7.5 Polymorphism in Inheritance :

- Polymorphism is widely used in inheritance. Inheritance is the process of deriving a new from already existing class. When a class is derived from base class all the public and pro attributes and methods of base class are also derived in derived class. Sometimes it is req to change definition of the method inherited from base class according to the requirement of de class. This can be done using the concept of **Method Overriding** in Python.
- Method Overriding is the concept of re defining method of base class in derived class. We only redefine implementation of the method in derived class. We cannot redefine number of parame and sequence of parameters in method definition.
- Consider Following Example :

```
class Shape:
    def PropertyOfShape(self):
        print("Different Shape Have Different Properties")
class Square(Shape):
    def __init__(self,Name):
        self.Name=Name
    def PropertyOfShape(self):
        print("This Is {}".format(self.Name))
        print("{} has 4 Sides".format(self.Name))
class Triangle(Shape):
    def __init__(self,Name):
        self.Name=Name
    def PropertyOfShape(self):
        print("This Is {}".format(self.Name))
        print("{} has 3 Sides".format(self.Name))
S1 = Shape()
S1.PropertyOfShape()
S2 = Square("Square")
S2.PropertyOfShape()
S3 = Triangle("Triangle")
S3.PropertyOfShape()
```

- In above example **Shape** is the Base class from which two classes named **Square** and Trian are derived. Shape class has one public method named **PropertyOfShape()**. As Square class Triangle class are derived from Shape class method of Shape class is also derived in Square c and Triangle class. In both Square class and Triangle class we redefine method PropertyOfSha as per the requirement. If you invoke method using an object of base class than it will exe base version of method and if you invoke method using an object of derived class than it execute derived version of method.

2.8 Abstraction - Abstract class :

Imp

- Data abstraction refers to the process of providing only essential information to the outside word and hiding their background details. Thus only that information is represented that is needed without explaining the details.
- For Example consider a TV in which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players BUT you do not know it's internal detail that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.
- Thus we can say, a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.
- Now if we talk in terms of Python Programming, Python classes provides great level of **data abstraction.** They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data i.e. State without actually knowing how class has been implemented internally.

2.8.1 Abstract Class :

- A class with one or more abstract methods defined inside it is known as **Abstract Class.** A method which is defined in abstract class but does not have its own implementation code in that class is called **Abstract Method.** In Object Oriented Programming Abstract Class acts as a blueprint for derived class. It contains set of methods which are common for derived classes but the implementation of method is different for different derived classes. Derived class must have to implement all abstract methods of Abstract Class. We cannot create an instance of Abstract class.
- Python itself does not provide the abstract class. In order to use Abstract class we need to import the module named **abc**, which provides the base for defining Abstract Base classes (ABC). We use the ***@abstractmethod*** decorator to define a method as abstract method. We can also define an abstract method by simply writing **pass** in the implementation of method. pass indicates that method does not have its own implementation.
- Consider Following Example :

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def Area(self):
        pass
class Square(Shape):
    def __init__(self,l):
        self.l=l
    def Area(self):
        return self.l**2
class Rectangle(Shape):
    def __init__(self,l,b):
        self.l=l
        self.b=b
    def Area(self):
```