CHAPTER

10 Inheritance

- Inheritance and its Types • Method Overriding • Containership
- Abstract Class and Interface • Metaclass

10.1 INTRODUCTION

Reusability is an important feature of OOP. Reusing an existing piece of code has manifolds benefits. It not only saves effort and cost required to build a software product, but also enhances its reliability. Now, no longer it will be required to re-write, re-debug, and re-test the code that has already been tested and being used in existing software.

To support reusability, Python supports the concept of re-using existing classes. For this, Python allows its programmers to create new classes that re-use the pre-written and tested classes. The existing classes are adapted as per user's requirements so that the newly formed classes can be incorporated in current software application being developed.

The technique of creating a new class from an existing class is called *inheritance*. The old or existing class is called the *base class* and the new class is known as the *derived class* or *subclass*. The derived classes are created by first inheriting the data and methods of the base class and then adding new specialized data and functions in it. In this process of inheritance, the base class remains unchanged. The concept of inheritance is therefore, frequently used to implement the `'is-a'` relationship. For example, teacher `is-a` person, student `is-a` person; while both teacher and student are a person in the first place, both also have some distinguishing features. So all the common traits of teacher and student are specified in the `Person` class and specialized features are incorporated in two separate classes—`Teacher` and `Student`. Similarly, a dentist or a surgeon is a doctor and doctor is a person. Figure 10.1 illustrates the concept of inheritance which follows a top–down approach to problem solving. In ***top-down*** approach, generalized classes are designed first and then specialized classes are derived by inheriting/extending the generalized classes.

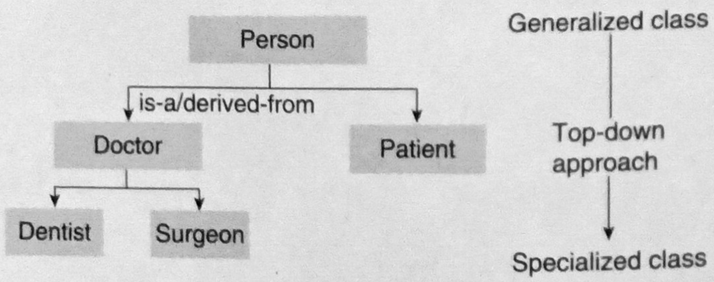

Figure 10.1 is-a relationship between classes

Note The derived class inherits all the capabilities of the base class and adds refinements and extensions of its own.

Remember that, when we make functions, we first write the individual functions and then call them from our main module. So we are building our program using small-small pieces (individual functions). This approach is called ***bottom-up approach***. But in case of class hierarchy, we are first designing base classes and then from those classes, specialized classes are created as we go down the hierarchy. This is a top-down approach. In bottom-up approach, the main deliverable is at the top of the hierarchy but in a top-down approach, the main deliverable is at the bottom.

10.2 INHERITING CLASSES IN PYTHON

The syntax to inherit a class can be given as,

```
class DerivedClass(BaseClass):
    body_of_derived_class
```

Note that instead of writing the `BaseClass`, you can even specify an expression like `modulename.BaseClass`. This is especially useful when the base class is defined in a different module. Let us look at the following example.

Example 10.1 Program to demonstrate the use of inheritance

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print("NAME : ", self.name)
        print("AGE : ", self.age)
class Teacher(Person):
    def __init__(self, name, age, exp, r_area):
        Person.__init__(self, name, age)
        self.exp = exp
        self.r_area = r_area
    def displayData(self):
        Person.display(self)
        print("EXPERIENCE : ", self.exp)
        print("RESEARCH AREA : ", self.r_area)
class Student(Person):
    def __init__(self, name, age, course, marks):
        Person.__init__(self, name, age)
        self.course = course
        self.marks = marks
    def displayData(self):
        Person.display(self)
        print("COURSE : ", self.course)
        print("MARKS : ", self.marks)
```

```
print("*********TEACHER***********")
T = Teacher("Jaya", 43, 20, "Recommender Systems")
T.displayData()
print("*********STUDENT***********")
S = Student("Mani", 20, "BTech", 78)
S.displayData()
```

OUTPUT

```
*********TEACHER***********
NAME :  Jaya
AGE :  43
EXPERIENCE :  20
RESEARCH AREA :  Recommender Systems
*********STUDENT***********
NAME :  Mani
AGE :  20
COURSE :  BTech
MARKS :  78
```

In the aforementioned program, classes `Teacher` and `Student` are both inherited from class `Person`. Therefore, the inherited classes have all the features (attributes and methods) of the base class. Note that a derived class is instantiated in the same way as any other class is. To create an object of the derived class, just write the derived class name followed by an empty brackets as in `DerivedClassName()`.

Note When we use the `.__base__` attribute with class name, the base (or the parent) class of the specified class is returned. Therefore, `print(Student.__bases__)` will print `(<class '__main__.Person'>,)`

10.2.1 Polymorphism and Method Overriding

Polymorphism, in simple terms, refers to having several different forms. It is one of the key features of OOP. It enables the programmers to assign a different meaning or usage to a variable, function, or an object in different contexts. While inheritance is related to classes and their hierarchy, polymorphism, on the other hand, is related to methods. When polymorphism is applied to a function or method depending on the given parameters, a particular form of the function can be selected for execution. In Python, method overriding is one way of implementing polymorphism.

Relationship Between Inheritance and Polymorphism

Polymorphism, an essential concept of OOP, means having several different forms. While inheritance is related to classes and their hierarchy, polymorphism, on the other hand, is related to methods. Polymorphism allows the programmers to assign a different meaning or usage to a method in different contexts. In Python, the word Polymorphism when used with inheritance means defining a number of subclasses that have methods of same name. A function can use objects of any of the polymorphic classes irrespective of the fact that these classes are individually distinct. Thus, in Python, one way of providing polymorphism is method overriding in which a derived class method has methods of same name as specified in the base class but giving it a new meaning.

In the program (Example 10.1) given under Section 10.2, notice that `__init__()` method was defined in all the three classes. When this happens, the method in the derived class overrides that in the base class. This means that `__init__()` in `Teacher` and `Student` gets preference over the `__init__()` method in the `Person` class. Thus, ***method overriding*** is the ability of a class to change the implementation of a method provided by one of its ancestors. It is an important concept of OOP since it exploits the power of inheritance.

Observe another thing that when we override a base class method, we extend the functionality of the base class method. This is done by calling the method in the base class method from the derived class method and also adding additional statements in the derived class method.

Instead of writing `Person.__init__(self, name, age)`, you could have also written `super().__init__(self, name, age)`. Here, `super()` is a built-in function that denotes the base class. So when you invoke a method using the `super()` function, then the parent version of the method is called.

Note In Python, every class is inherited from the base class object.

Note that in case of multiple inheritance (a class derived from more than one base class), you need to invoke the `super()` function in `__init__()` method of every class. This would be clear by looking at the program given below and observing its output.

Example 10.2 Program to demonstrate the issue of invoking `__init__()` in case of multiple inheritance.

```
class Base1(object):
    def __init__(self):
        print("Base1 Class")
class Base2(object):
    def __init__(self):
        print("Base2 Class")
class Derived(Base1, Base2):
    pass
D = Derived()
```

OUTPUT

```
Base1 Class
```

In the above method, an object of derived class is made. Since there is no `__init__()` method in the derived class, the `__init__()` method of the first base class gets called. But since, there is no call to `super()` function in the `__init__()` method of `Base1` class, no further `__init__()` method is invoked. This problem has been recitified in the code given in the following example.

Example 10.3 Program to demonstrate the call of `super()` from `__init__()` of a base class

```
class Base1(object):
    def __init__(self):
        print("Base1 Class")
        super(Base1, self).__init__()
class Base2(object):
    def __init__(self):
```

```
        print("Base2 Class")
class Derived(Base1, Base2):
    pass
D = Derived()
```

OUTPUT

```
Base1 Class
Base2 Class
```

Example 10.4 Program to call the `__init__()` methods of all the classes

```
class Base1(object):
    def __init__(self):
        print("Base1 Class")
        super(Base1, self).__init__()
class Base2(object):
    def __init__(self):
        print("Base2 Class")
class Derived(Base1, Base2):
    def __init__(self):
        super(Derived, self).__init__()
        print("Derived Class")
D = Derived()
```

OUTPUT

```
Base1 Class
Base2 Class
Derived Class
```

Two more built-in functions `isinstance()` and `issubclass()` are very useful in Python to check instances. The `isinstance()` function returns `True` if the object is an instance of the class or other classes derived from it. Similarly, the `issubclass()` checks for class inheritance as shown in the following example. Just try the following statements and observe the output.

Example 10.5 Program to demonstrate `isinstance()` and `issubclass()`. (Note that the following code is in continuation to Example 10.1 where we had defined classes—`Person`, `Teacher`, and `Student`).

```
print("T is a Teacher : ", isinstance(T,Teacher))
print("T is a Person : ", isinstance(T,Person))
print("T is an integer : ", isinstance(T,int))
print("T is an object : ", isinstance(T,object))
print("Person is a subclass of Teacher : ", issubclass(Person,Teacher))
print("Teacher is a subclass of Person : ", issubclass(Teacher,Person))
print("Boolean is a subclass of int : ", issubclass(bool,int))
```


OUTPUT

```
T is a Teacher :  True
T is a Person :  True
T is an integer :  False
T is an object :  True
Person is a subclass of Teacher :  False
Teacher is a subclass of Person :  True
Boolean is a subclass of int :  True
```

10.3 TYPES OF INHERITANCE

Python supports different variants of inheritance such as single, multiple, multi-level, and multi-path inheritances. While, in single inheritance, a class can be derived from a single base class, in multiple inheritance, on the other hand, a class can be derived from more than one base class. Besides these, Python has other types of inheritance which will be discussed in this section.

10.3.1 Multiple Inheritance

When a derived class inherits features from more than one base class (Figure 10.2), it is called *multiple inheritance*. The derived class has all the features of both the base classes and in addition to them, can have additional new features. The syntax for multiple inheritance is similar to that of single inheritance and can be given as:

```
class Base1:
    statement block
class Base2:
    statement block
class Derived(Base1, Base2):
    statement block
```

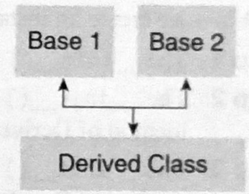

Figure 10.2 Multiple inheritance

In the multiple inheritance scenario, any specified attribute is first searched in the current (or the derived) class. If it is not found there, the search continues into parent classes using depth-first technique, that is, in left-right fashion without searching same class twice. Let us take an example to better understand this concept.

Note If the specified attribute is not found in the derived class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

Example 10.6 Program to demonstrate multiple inheritance

```
class Base1(object):        # First Base Class
  def __init__(self):
    super(Base1, self).__init__()
    print("Base1 Class")
class Base2(object):        # Second Base Class
  def __init__(self):
```

```
    super(Base2, self).__init__()
    print("Base2 Class")
class Derived(Base1, Base2):    # Derived Class derived from Base1 and Base2
  def __init__(self):
    super(Derived, self).__init__()
    print("Derived Class")
D = Derived()
```

OUTPUT

```
Base2 Class
Base1 Class
Derived Class
```

The order of output may confuse you. But do not worry, it's all because of MRO (that works on depth-first traversal) which will be discussed shortly. For now, just understand that the order of class hierarchy can be given as—Derived -> Base1 -> Object and Derived -> Base2 -> Object.

When we create an instance of the derived class, the following things happen.

Step 1: The `__init__()` method of Derived class is called.

Step 2: The `__init__()` method of Base1 class is invoked (according to MRO) from the `__init__()` method of Derived class.

Step 3: The `__init__()` method of Base2 class is invoked (according to MRO) from the `__init__()` method of Base1 class.

Step 4: From the `__init__()` method of Base2, the `__init__()` method of Object is invoked which does nothing. Finally, Base2 class gets printed on the screen and the control is returned to the `__init__()` method of Base1 class.

Step 5: Base1 class gets printed and the control is transferred back to the `__init__()` method of Derived class.

Step 6: Derived class gets printed on the screen and hence the result.

10.3.2 Multi-level Inheritance

The technique of deriving a class from an already derived class is called *multi-level inheritance*. In Figure 10.3, Base Class acts as the base for *Derived Class 1* which in turn acts as a base for *Derived Class 2*. The *Derived Class 1* has features of *Base Class* plus its own features. The *Derived Class 1* is known as the *intermediate base class* as this class provides a link for inheritance between the *Base Class* and the *Derived Class 2*. The chain of classes—`Base Class -> Derived Class 1 -> Derived Class 2` is known as the *inheritance path*. In multi-level inheritance, number of levels can go up to any number based on the requirement. The syntax for multi-level inheritance can be given as,

```
class Base:
    pass
class Derived1(Base):
    pass
class Derived2(Derived1):
    Pass
```

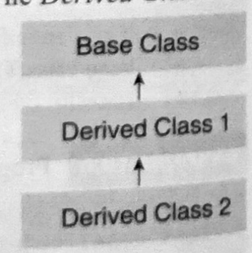

Figure 10.3 Multi-level inheritance

In multi-level inheritance scenario, any specified attribute is first searched in the current class (*Derived Class 2*). If it is not found there, then the *Derived Class 1* is searched, if it is not found even there then the *Base Class* is searched. If the attribute is still not found, then finally the object class is checked. This order is also called *linearization* of *Derived Class 2*. Correspondingly, the set of rules used to find this linearization order is called *Method Resolution Order (MRO)*.

Programming Tip: All methods in Python are effectively virtual.

The MRO ensures that a class appears before its parent classes. However, in case of multiple inheritance, the MRO is the same as a tuple of base classes. You can check the MRO of a class by either using the `__mro__` attribute or the `mro()` method. While the `__mro__` attribute returns a tuple, the `mro()` method returns a list.

Note Python has MRO and an algorithm C3 to keep a track of classes and their hierarchy.

Example 10.7 Program to demonstrate multi-level inheritance

```
class Person:                    # Base class
   def name(self):
     print('Name...')
class Teacher(Person):           # Class derived from Person
  def Qualification(self):
     print('Qualification...Ph.D must')
class HOD(Teacher):    # Class derived from Teacher, now hierarchy is Person-
>Teacher->HOD
   def experience(self):
       print('Experience......at least 15 years')
hod = HOD()
hod.name()
hod.Qualification()
hod.experience()
```

OUTPUT

```
Name...
Qualification...Ph.D must
Experience......at least 15 years
```

10.3.3 Multi-path Inheritance

Deriving a class from other derived classes that are in turn derived from the same base class is called *multi-path inheritance*. As seen in the Figure 10.4, the derived class has two immediate base classes— *Derived Class 1* and *Derived Class 2*. Both these base classes are themselves derived from the *Base Class*, thereby forming a grandparent, parent, and child form of a relationship. The derived class inherits the features of the *Base Class* (grandparent) via two separate paths. Therefore, the *Base Class* is also known as the *indirect base class*.

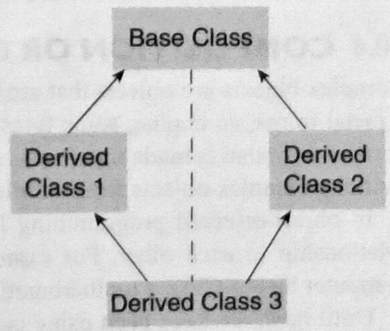

Figure 10.4 Multi-path inheritance

Example 10.8 Program to demonstrate multi-path inheritance

```
class Student:
    def name(self):
      print('Name...')
class Academic_Performance(Student):
   def Acad_score(self):
      print('Academic Score...90% and above')
class ECA(Student):
    def ECA_score(self):
        print('ECA Score......60% and above')
class Result(Academic_Performance, ECA):
    def Eligibility(self):
        print("*******Minimum Eligibility to Apply*******")
        self.Acad_score()
        self.ECA_score()

R = Result()
R. Eligibility()
```

OUTPUT

```
*******Minimum Eligibility to Apply*******
Academic Score...90% and above
ECA Score......60% and above
```

Problem in Multi-Path Inheritance (Diamond Problem)

The derived class inherits the members of the base class (grandparent) twice, via parent1 (*Derived Class 1*) and via parent 2 (*Derived Class 2*). This results in ambiguity because a duplicate set of members is created. This ambiguous situation must be avoided.

Thus, we see that diamond relationships exist when at least one of the parent classes can be accessed through multiple paths from the bottommost class. Diamond relationship is very common in Python as all classes inherit from the object and in case of multiple inheritance there is more than one path to reach the object. To prevent base classes from being accessed more than once, the dynamic algorithm (C3 and the MRO) linearizes the search order in such a way that the left-to-right ordering specified in each class is preserved and each parent is called only once (also known as monotonic).

10.4 COMPOSITION OR CONTAINERSHIP OR COMPLEX OBJECTS

Complex objects are objects that are built from smaller or simpler objects. For example, a car is built using a metal frame, an engine, some tyres, a transmission, a steering wheel, and several other parts. Similarly, a computer system is made up of several parts such as CPU, motherboard, memory, and so on. This process of building complex objects from simpler ones is called *composition or containership.*

In object-oriented programming languages, object composition is used for objects that have a has-a relationship to each other. For example, a car has-a metal frame, has-an engine, etc., and a personal computer has-a CPU, a motherboard, and other components.

Until now, we have been using classes that have data members of built-in type. While this worked well for simple classes, for designing classes that simulate real world applications, programmers often need data members that belong to other simpler classes.

Remember that in *composition,* the two objects are quite strongly linked. This means that one object can be thought of as exclusively *belonging* to the other object. If the owner object ceases to exist, the owned object will also cease to exist.

Note In composition, complex classes have data members belonging to other simpler classes.

Benefits

- Each individual class can be simple and straightforward.
- A class can focus on performing one specific task.
- The class is easier to write, debug, understand, and be usable by other programmers.
- While simpler classes can perform all the operations, the complex class can be designed to coordinate the data flow between simpler classes.
- It lowers the overall complexity of the complex object because the main task of the complex object would then be to delegate tasks to the sub-objects, who already know how to do them.

Scope of Use

Although there is no well-defined rule to state when a programmer must use composition, as a rule of thumb, each class should be built to accomplish a single task. The task should be to either perform some part of manipulation or be responsible for coordinating other classes but cannot perform both tasks. Following are some points which you should remember while deciding whether to use composition or inheritance.

- Try to limit the use of multiple inheritance as it makes the program complex to read, underst, and and debug.
- Composition should be used to package code into modules that are used in many different unrelated pieces of codes.
- Inheritance should be used only when there are clearly related reusable pieces of code that fits under a single common concept or if you are specifically asked to use it.

Note If the link between two objects is weak, and neither object has exclusive ownership of the other, then it is not composition. It is rather called *aggregation.*

Example 10.9 Program that uses complex objects

```
class One:
    def set(self,var):
        self.var = var
    def get(self):
        return self.var
class Two:
    def __init__(self, var):
        self.o = One()    # object of class One is created
    # method of class One is invoked using its object in class Two
        self.o.set(var)
    def show(self):
        print("Number = ", self.o.get())
T = Two(100)
T.show()
```

OUTPUT

```
Number = 100
```

Note that in the aforementioned program, class Two has an object of class One as its data member. To access a member of One, we must use objects of both the classes as in `self.o.get()`. Thus, we see that composition is generally used when the features of an existing class is needed inside the new class, but not its interface. For this, existing class's object is embedded in the new class. The programmer will use the interface of the new class but implementation details of the original class.

A comparison between containership and inheritance is given in Table 10.1.

Table 10.1 Comparison between Inheritance and Containership

Inheritance	Containership
• Enables a class to inherit data and functions from a base class by extending it.	• Enables a class to contain objects of different classes as its data member.
• The derived class may override the functionality of base class.	• The container class cannot alter or override the functionality of the contained class.
• The derived class may add data or functions to the base class.	• The container class cannot add anything to the contained class.
• Inheritance represents a "is-a" relationship.	• Containership represents a "has-a" relationship.
• Example: A Student is a Person.	• Example: class One has a class Two.

10.5 ABSTRACT CLASSES AND INTERFACES

In some OOP languages like C++ and Java, it is possible to create a class which cannot be instantiated. This means that that you cannot create objects of that class. Such classes could only be inherited and then an object of the derived class was used to access the features of the base class. Such a class was known as the abstract class.

An abstract class corresponds to an abstract concept. For example, a polygon may refer to a rectangle, triangle, or any other closed figure. Therefore, an abstract class is a class that is specifically defined to lay a foundation for other classes that exhibits a common behaviour or similar characteristics. It is primarily used only as a base class for inheritance.

Programming Tip: Instantiating an object of an abstract class causes an error.

Since an abstract class is an incomplete class, users are not allowed to create its object. To use such a class, programmers must derive it keeping in mind that they would only be either using or overriding the features specified in that class.

Therefore, we see that an abstract class just serves as a *template* for other classes by defining a list of methods that the classes must implement. It makes no sense to instantiate an abstract class because all the method definitions are empty and must be implemented in a subclass.

The abstract class is thus an *interface* definition. In inheritance, we say that a class *implements* an interface if it inherits from the class which specifies that interface. In Python, we use the `NotImplementedError` to restrict the instantiation of a class. Any class that has the `NotImplementedError` inside method definitions cannot be instantiated. Consider the program given in the following example which creates an abstract class `Fruit`. Two other classes, `Mango` and `Orange` are derived from `Fruit` that implements all the methods defined in `Fruit`. Then we create the objects of the derived classes to access the methods defined in these classes.

Example 10.10 Program to illustrate the concept of abstract class

```
class Fruit:
    def taste(self):
        raise NotImplementedError()
    def rich_in(self):
        raise NotImplementedError()
    def colour(self):
        raise NotImplementedError()
class Mango(Fruit):
    def taste(self):
        return "Sweet"
    def rich_in(self):
        return "Vitamin A"
    def colour(self):
        return "Yellow"
class Orange(Fruit):
    def taste(self):
        return "Sour"
    def rich_in(self):
        return "Vitamin C"
    def colour(self):
        return "Orange"
Alphanso = Mango()
print(Alphanso.taste(), Alphanso.rich_in(), Alphanso.colour())
Org = Orange()
print(Org.taste(), Org.rich_in(), Org.colour())
```

OUTPUT

```
Sweet Vitamin A Yellow
Sour Vitamin C Orange
```

Programming Tip: Super falls apart if the methods of subclasses do not take the same arguments.

10.6 METACLASS

A metaclass is the class of a class. While a class defines how an instance of the class behaves, a metaclass, on the other hand, defines how a class behaves. Every class that we create in Python is an instance of a metaclass (refer Figure 10.5).

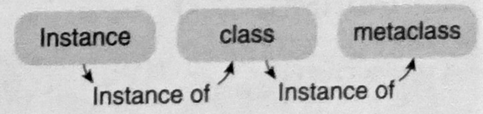

Figure 10.5 Concept of metaclass

For example, *type* is a metaclass in Python. It is itself a class, and it is its own type. Although, you cannot make an exact replica of something like type, but Python does allow you to create a metaclass by making a subclass type.

A ***metaclass*** is most commonly used as a class factory. As we create an instance of the class by calling the class, Python creates a new class by calling the metaclass. By defining `__init__()` and `__new__()` methods in the metaclass, you can do a lot of extra things (while creating a class) like registering the new class with some registry, or replacing the class completely.

Programming Tip: In Python, a new class is created by calling the metaclass.

Python allows you to define normal methods on the metaclass which are like classmethods, as they can be called on the class without an instance. However, there is a difference between them as that they cannot be called

on an instance of the class. Python also allows you to define the normal magic methods, such as `__add__()`, `__iter__()`, and `__getattr__()`, to implement or change how the class behaves.

Substitutability

Substitutability is a principle in object oriented programming that states that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S. In other words, an object of type T may be substituted with an object of a subtype S without changing any desirable property of type T.

Liskov has given a *substitution* principle also known as the **Liskov substitution principle (LSP)** which defines a subtyping relation, called **(strong) behavioural subtyping**. It is a semantic rather than syntactic relation.

PROGRAMMING EXAMPLES

Program 10.1 Write a program that has a class Point. Define another class Location which has two objects (Location and Destination) of class Point. Also define a function in Location that prints the reflection of Destination on the x axis.

```
class Point:
    def __init__(self, x, y):
        self. x = x
        self.y = y
    def get(self):
        return (self.x,self.y)
class Location:
    def __init__(self,x1,y1,x2,y2):
        self.Source = Point(x1,y1)
        self.Destination = Point(x2,y2)
    def show(self):
        print("Source = ", self.Source.get())
        print("Destination = ", self.Destination.get())
    def reflection(self):
        self.Destination.x = -self.Destination.x
        print("Reflection Point on x Axis is : ", self.Destination.x, self.
Destination.y)
L = Location(1,2,3,4)
L.show()
L.reflection()
```

OUTPUT

```
Source =  (1, 2)
Destination =  (3, 4)
Reflection Point on x Axis is :  -3 4
```

Program 10.2 Write a program that has classes such as Student, Course, and Department. Enroll a student in a course of a particular department.

```
class Student:
    def __init__(self, name, rollno, course, year):
```



```
        self.name = name
        self.rollno = rollno
        self.course = Course(course, year)
    def show(self):
        print(self.name, self.rollno)
        print(self.course.get())
class Course:
    def __init__(self, name, year):
        self.name = name
        self.year = year
    def get(self):
        return(self.name, self.year)
class Deptt:
    def __init__(self, name):
        self.name = name
        self.courses = []
    def get(self):
        return(name, courses)
    def add_courses(self,name):
        self.courses.append(name)
    def show_courses(self):
        print("Courses offered in this department are : ",self.courses)
D1 = Deptt("Mathematics")
D2 = Deptt("Computer Science")
D1.add_courses("BA(H)")
D1.add_courses("BSc(H)")
D2.add_courses("BCA")
D2.add_courses("BTech")
print("*** Dear Students, the list of courses offered in their respective
departments is given below.. Kindly choose any one course*************")
D1.show_courses()
D2.show_courses()
S = Student("Harman", 1234, "BCA", 2017)
S.show()
```

OUTPUT

```
*** Dear Students, the list of courses offered in their respective departments is
given below.. Kindly choose any one course*************
Courses offered in this department are :  ['BA(H)', 'BSc(H)']
Courses offered in this department are :  ['BCA', 'BTech']
Harman 1234
('BCA', 2017)
```

Program 10.3 Write a program that has an abstract class Polygon. Derive two classes Rectangle and Triangle from Polygon and write methods to get the details of their dimensions and hence calculate the area.

```
class Polygon:
    def get_data(self):
        raise NotImplementedError()
    def area(self):
        raise NotImplementedError()
class Rectangle(Polygon):
    def get_data(self):
        self.length = float(input("Enter the Length of the Rectangle : "))
        self.breadth = float(input("Enter the Breadth of the Rectangle : "))
    def area(self):
        return self.length * self.breadth;
class Triangle(Polygon):
    def get_data(self):
        self.base = float(input("Enter the Base of the Triangle : "))
        self.height = float(input("Enter the Height of the Triangle : "))
    def area(self):
        return 0.5*self.base * self.height;
R = Rectangle()
R.get_data()
print("Area of Rectangle : ", R.area())
T = Triangle()
T.get_data()
print("Area of Triangle : ", T.area())
```

OUTPUT

```
Enter the Length of the Rectangle : 70
Enter the Breadth of the Rectangle : 30
Area of Rectangle :  2100.0
Enter the Base of the Triangle : 50
Enter the Height of the Triangle : 100
Area of Triangle :  2500.0
```

Program 10.4 Write a program with class Bill. The users have the option to pay the bill either by cheque or by cash. Use the inheritance to model this situation.

```
class Bill:
    def __init__(self, items, price):
        self.total = 0;
        self.items = items
        self.price = price
        for i in self.price:
            self.total += i
    def display(self):
        print("\n ITEM \t\t\t PRICE")
        for i in range(len(self.items)):
                print(self.items[i], "\t", self.price[i])
        print("************************")
        print("TOTAL = ", self.total)
class Cash_Payment(Bill):
    def __init__(self, items,  price, deno, value) :
        Bill.__init__(self,items, price)
        #self.n = n;
        self.deno = deno
        self.value = value
    def show_Cash_Payment_Details(self):
        Bill.display(self)
        for i in range(len(deno)):
            print(deno[i], "*", value[i], " = ", deno[i] * value[i])
class Cheque_Payment(Bill):
    def __init__(self, items, price, cno, name) :
        Bill.__init__(self, items, price)
        self. cno = cno
        self.name = name
    def show_Check_Payment_Details(self):
        Bill.display(self)
        print("CHEQUE NUMBER : ", self.cno)
        print("BANK NAME : ", self.name)
items = ["External Hard Disk", "RAM", "Printer", "Pen Drive"]
price = [5000, 2000, 6000, 800]
option = int(input("Would you like to pay by cheque or cash (1/2): "))
if(option==1):
    name = input("Enter the name of the bank : ")
    cno = input("Enter the cheque number : ")
    Cheque = Cheque_Payment(items, price, cno, name)
    Cheque.show_Check_Payment_Details()
else:
    deno = [10, 20,50, 100, 500, 2000]
    value = [1,1,1,20,4,5]
    CP = Cash_Payment(items, price, deno, value)

    CP.show_Cash_Payment_Details()
```

OUTPUT

```
Would you like to pay by cheque or cash (1/2): 1
Enter the name of the bank : ICICI
Enter the cheque number : 12345
ITEM                PRICE
External Hard Disk  5000
RAM                 2000
Printer             6000
Pen Drive           800
************************
TOTAL =  13800
```

```
CHEQUE NUMBER :  12345
BANK NAME :  ICICI
```

Program 10.5 Write a program that has a class Person. Inherit a class Faculty from Person which also has a class Publications.

```
class Person:
    def __init__(self, name, age, sex):
        self.name = name
        self.age = age
        self.sex = sex
    def display(self):
        print("NAME : ", self.name)
        print("AGE : ", self.age)
        print("SEX : ", self.sex)
class Publications:
    def __init__(self, no_RP, no_Books, no_Art):
        self.no_RP = no_RP
        self.no_Books = no_Books
        self.no_Art = no_Art
    def display(self):
        print("Number of Research papers Published : ", self.no_RP)
        print("Number of Books Published : ", self.no_Books)
        print("Number of Articles Published : ", self.no_Art)
class Faculty(Person):
    def __init__(self, name, age, sex, desig, dept,no_RP, no_Books, no_Art):
        Person.__init__(self, name, age, sex)
        self.desig = desig
        self.dept = dept
        self.Pub = Publications(no_RP, no_Books, no_Art)
    def display(self):
        Person.display(self)
        print("DESIGNATION : ", self.desig)
        print("DEPARTMENT : ", self.dept)
        self.Pub.display()
F = Faculty("Pooja", 38, "Female", "TIC", "Computer Science", 22, 1, 3)
F.display()
```

OUTPUT

```
NAME :  Pooja
AGE :  38
SEX :  Female
DESIGNATION :  TIC
DEPARTMENT :  Computer Science
Number of Research papers Published :  22
Number of Books Published :  1
Number of Articles Published :  3
```