3	❖ **Classes, Inheritance and Polymorphism**
	❖ **Classes** Creating a class, The Self variable, Constructor, Types of variables, Types of methods, Passing members of one class to another.
	❖ **Inheritance** Implementing inheritance, Constructors in inheritance, Overriding Super class constructors and methods, The super() method, Types of Inheritance, Single and multiple, problems in multiple inheritance, Method resolution order(MRO).
	❖ **Polymorphism** Introduction to polymorphism, Duck Typing Philosophy of Python, Operator overloading, method overloading, method overriding.

CHAPTER

12

INTRODUCTION TO OOPS

In this chapter, we will introduce a revolutionary concept called *Object Oriented Programming System* (OOPS) based on which the languages like Smalltalk, Simula-67, C++, Java, Python, etc. are created. In this chapter, however, we are going to have a bird's eye view of the fundamental concepts of OOPS while an in depth discussion will be held only in the subsequent chapters.

The languages like C, Pascal, Fortran etc., are called Procedure Oriented Programming languages since in these languages, a programmer uses procedures or functions to perform a task. While developing software, the main task is divided into several sub tasks and each sub task is represented as a procedure or function. The main task is thus composed of several procedures and functions. This approach is called *Procedure oriented approach*. Consider Figure 12.1:

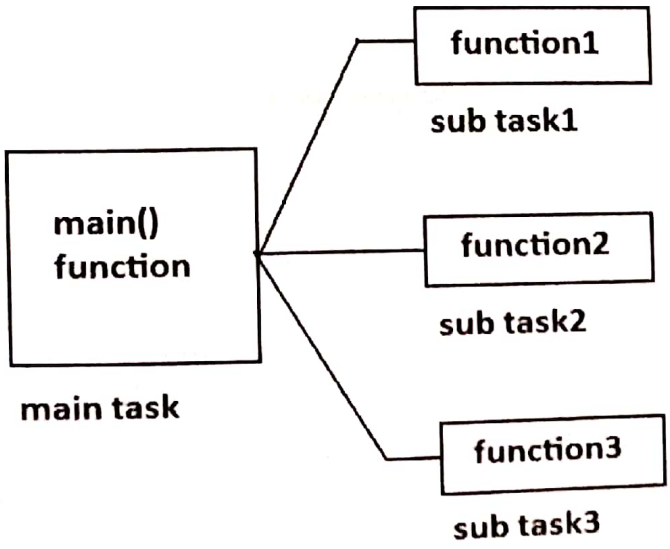

Figure 12.1: Procedure Oriented Approach

On the other hand, languages like C++, Java and Python use classes and objects in their programs and are called Object Oriented Programming languages. A class is a module which itself contains data and methods (functions) to achieve the task. The main task is divided into several sub tasks, and these are represented as classes. Each class can perform several inter-related tasks for which several methods are written in a class. This approach is called *Object Oriented approach*. Consider Figure 12.2:

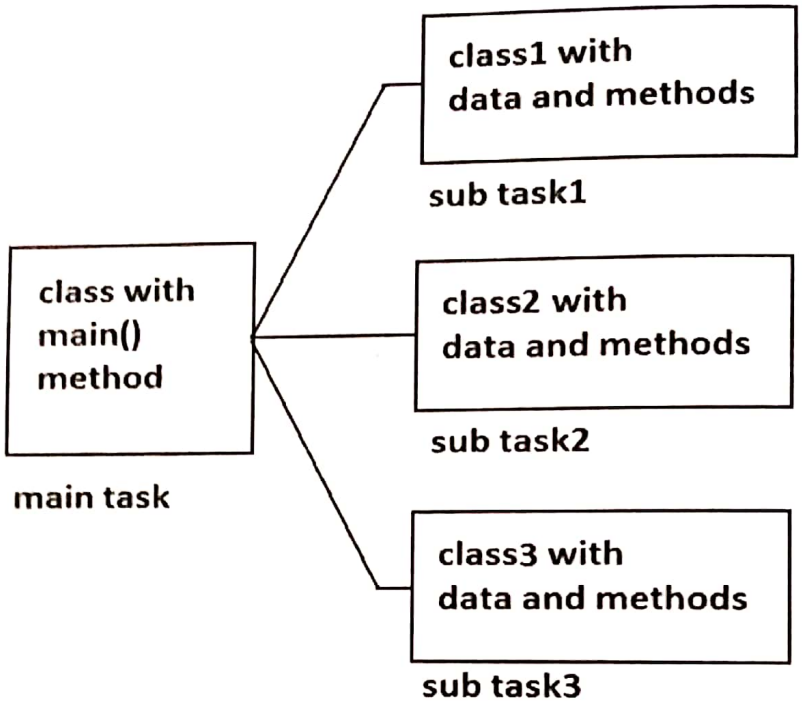

Figure 12.2: Object Oriented Approach

Programmers have followed Procedure Oriented approach for several decades, but as experience and observation teaches new lessons, programmers slowly realized several problems with Procedure Oriented approach.

Problems in Procedure Oriented Approach

In Procedure Oriented approach, the programmer concentrates on a specific task, and writes a set of functions to achieve it. When there is another task to be added to the software, he would be writing another set of functions. This means his concentration will be only on achieving the tasks. He perceives the entire system as fragments of several tasks. Whenever he wants to perform a new task, he would be writing a new set of functions. Thus there is no reusability of an already existing code. A new task every time requires developing the code from the scratch. This wastes programmer's time and effort.

In Procedure Oriented approach, every task and sub task is represented as a function and one function may depend on another function. Hence, an error in the software needs examination of all the functions. Thus debugging or removing errors will become difficult. Any updations to the software will also be difficult.

When the software is developed, naturally code size will also be increased. It has been observed in most of the software developed following the Procedure Oriented approach that when the code size exceeds 10,000 lines and before reaching 100,000 lines, suddenly at a particular point, the programmers start losing control on the code. This means, the programmers could not understand the exact behavior of the code and could neither debug it, nor extend it. This posed many problems, especially when the software was constructed to handle bigger and complex systems. For example, to create software to send satellites into the sky and control their operations from the ground stations, we may have to write millions of lines of code. In such systems, Procedure Oriented approach fails and programmers realized the need of another approach.

There is another problem with Procedure Oriented approach. Programming in this approach is not developed from human being's life. Statements, functions or procedures never reflect the human beings. So, from the human beings' point of view, they are unnatural. Unnatural activities are difficult to perform. For example, as human beings, we can walk or run. But, if we are asked to fly like birds; that would become impossible for us since flying is not natural for human beings. In the same way, if programming is developed from human beings' life, we can adapt to it easily. This idea forced computer scientists to develop a new approach that would be closer to human beings' life.

Due to the preceding reasons, computer scientists felt the need of a new approach where programming will have several modules. Each module represents a 'class' and the classes can be reusable and hence maintenance of code will become easy. When there is an error, it is possible to debug only on that class where error occurred without disturbing the other classes. This approach is suitable not only to develop bigger and complex applications but also to manage them easily. Moreover, this approach is built from a single root concept 'object', which represents anything that physically exists in this world. It means that all human beings are objects. All animals are objects. All existing things will become objects. This new approach is called 'Object Oriented Approach'. Programming in this approach is called Object Oriented Programming System (OOPS).

In OOPS, everything is an object. In real life, some objects will have similar behavior. For example, all birds have similar behavior like having two wings, two legs, etc. Also, all birds have the ability to fly in the sky. Such objects with similar behavior belong to the same class. So, a class represents common behavior of a group of objects. Since a class represents behavior, it does not exist physically. But objects exist physically. For example, bird is a class; whereas, sparrow, pigeon, crow and peacock are objects of the bird class. Similarly, human being is a class and Arjun, Krishna, Sita are objects of the human being class.

Specialty of Python Language

In OOPS, all programs involve creation of classes and objects. This makes programs lengthy. For example, we have to write all the statements of the program inside a class and then create objects to the class. Then use the features of the class through objects. This type of programming requires much code to perform a simple task like adding two numbers. Also, the program execution takes more time. So, for simple tasks, it is still better to go for procedure oriented approach which offers less code and more speed. For example, a C program executes faster than its equivalent program written in Python or Java!

Even though, Python is an object oriented programming language like Java, it does not force the programmers to write programs in complete object oriented way. Unlike Java, Python has a blend of both the object oriented and procedure oriented features. Hence, Python programmers can write programs using procedure oriented approach (like C) or object oriented approach (like Java) depending on their requirements. This is definitely an advantage for Python programmers!

Features of Object Oriented Programming System (OOPS)

There are five important features related to Object Oriented Programming System. They are:

- Classes and objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Let's move further to have clear understanding of each of these features.

Classes and Objects

The entire OOPS methodology has been derived from a single root concept called 'object'. An object is anything that really exists in the world and can be distinguished from others. This definition specifies that everything in this world is an object. For example, a table, a ball, a car, a dog, a person, etc. will come under objects. Then what is not an object? If something does not really exist, then it is not an object. For example, our thoughts, imagination, plans, ideas etc. are not objects, because they do not physically exist.

Every object has some behavior. The behavior of an object is represented by attributes and actions. For example, let's take a person whose name is 'Raju'. Raju is an object because he exists physically. He has attributes like name, age, sex, etc. These attributes can be represented by variables in our programming. For example, 'name' is a string type variable, 'age' is an integer type variable.

Similarly, Raju can perform some actions like talking, walking, eating and sleeping. We may not write code for such actions in programming. But, we can consider calculations and processing of data as actions. These actions are performed by methods. We should understand that a function written inside a class is called a method. So an object contains variables and methods.

It is possible that some objects may have similar behavior. Such objects belong to same category called a 'class'. For example, not only Raju, but all the other persons have various common attributes and actions. So they are all objects of same class, 'Person'. Now observe that the 'Person' will not exist physically but only Raju, Ravi, Sita, etc. exist physically. This means, a class is a group name and does not exist physically, but objects exist physically. See Figure 12.3:

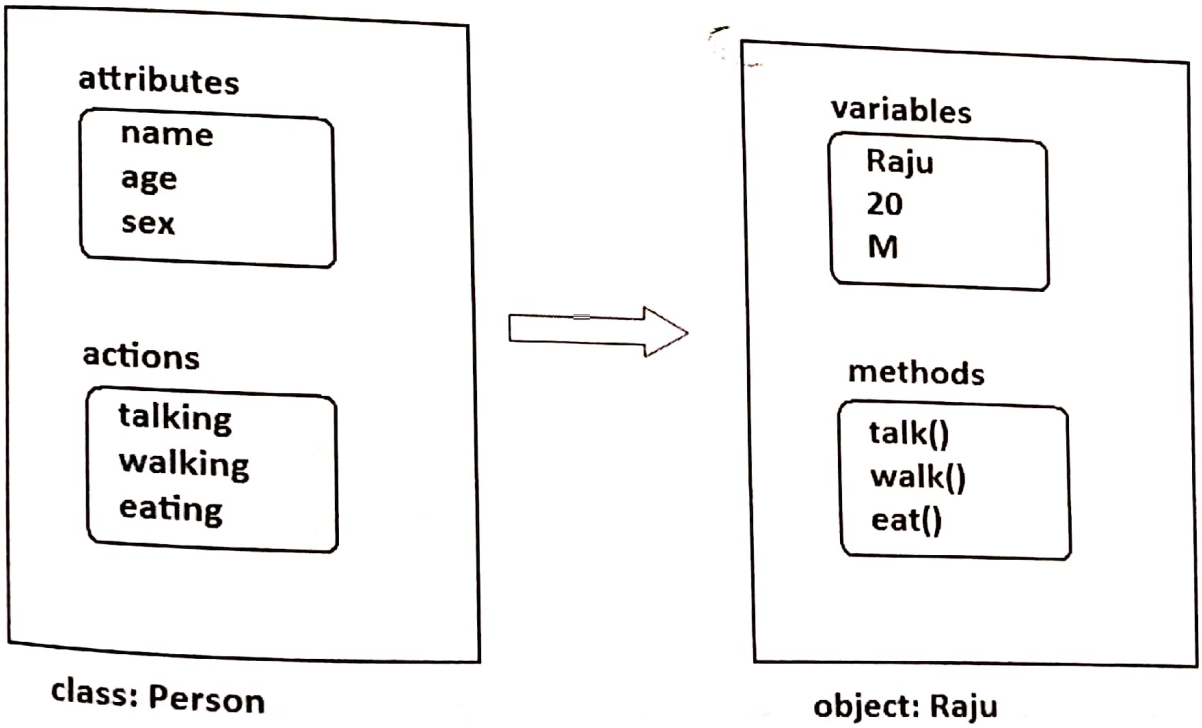

Figure 12.3: Person Class and Raju Object

To understand a class, take a pen and paper and write down all the attributes and actions of any person. The paper contains the model that depicts a person, so it is called a class. We can find a person with the name 'Raju', who got all the attributes and actions as written on the paper. So 'Raju' becomes an object of the class, Person. This gives a definition for the class. A class is a model or blueprint for creating objects. By following the class, one can create objects. So we can say, whatever is there in the class, will be seen in its objects also.

We can use a class as a model for creating objects. To write a class, we can write all the characteristics of objects which should follow the class. These characteristics will guide us to create objects. A class and its objects are almost the same with the difference that a class does not exist physically, while an object does. For example, let's say we want to construct a house. First of all, we will go to an architect who provides a plan. This plan is only an idea and exists on paper. This is called a class. However, based on this plan, if we construct the house, it is called an object since it exists physically. So, we can say that we can create objects from the class. An object does not exist without a class. But a class can exist without any objects.

Let's take another example. Flower is a class but if we take Rose, Lily, and Jasmine - they are all objects of flower class. The class flower does not exist physically but its objects, like Rose, Lily and Jasmine exist physically.

Let's take another example. We want a table made by a carpenter. First of all, the carpenter takes a paper and writes the measurements regarding length, breadth and height of the table. He may also draw a picture on the paper that works like a model for creating the original table. This plan or model is called a 'class'. Following this model, he makes the table that can be used by us. This table is called an 'object'. To make the table, we need material, i.e. wood. The wood represents the memory allotted by the PVM for the objects. Remember, objects are created on heap memory by PVM at run time. See Figure 12.4. It is also possible to create several objects (tables) from the same class (plan). An object cannot exist without a class. But a class can exist without any object. We can think that a class is a model and if it physically appears, then it becomes an object. So an object is called 'instance' (physical form) of a class.

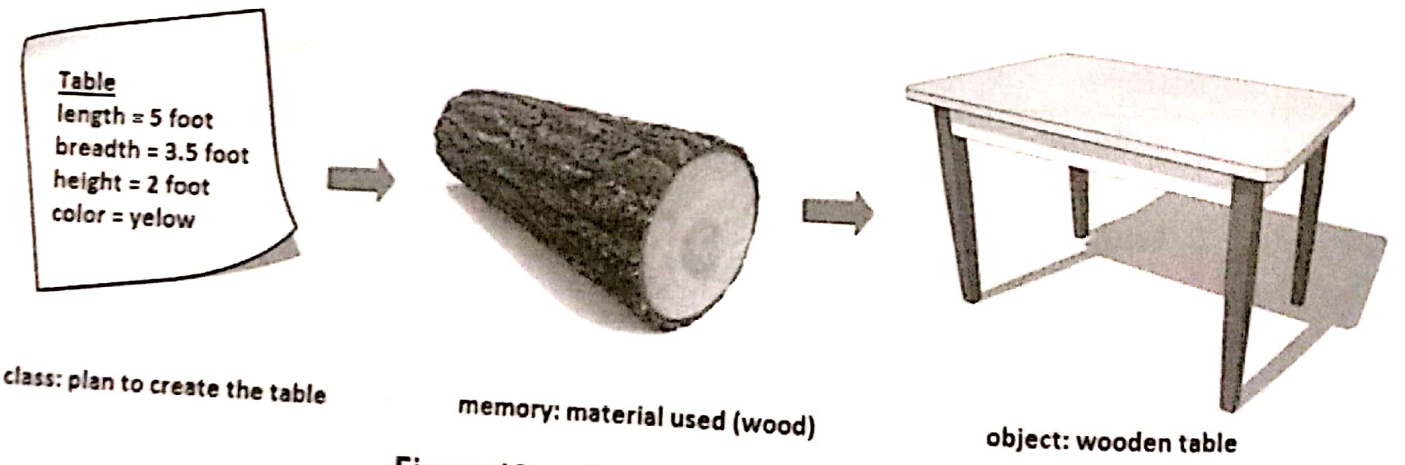

Figure 12.4: Creation of a Class and Object

Creating Classes and Objects in Python

Let's create a class with the name Person for which Raju and Sita are objects. A class is created by using the keyword, class. A class describes the attributes and actions performed by its objects. So, we write the attributes (variables) and actions (functions) in the class as:

```
# This is a class
class Person:
```

```
# attributes means variables
name = 'Raju'
age = 20

# actions means functions
def talk(cls):
    print(cls.name)
    print(cls.age)
```

Observe the preceding code. Person class has two variables and one function. The function that is written in the class is called method. When we want to use this class, we should create an object to the class as:

```
p1 = Person()
```

Here, p1 is an object of Person class. Object represents memory to store the actual data. The memory needed to create p1 object is provided by PVM. Observe the function (or method) in the class:

```
def talk(cls):
```

Here, 'cls' represents a default parameter that indicates the class. So, cls.name refers to class variable 'Raju'. We can call the talk() method to display Raju's details as:

```
p1.talk()
```

Encapsulation

Encapsulation is a mechanism where the data (variables) and the code (methods) that act on the data will bind together. For example, if we take a class, we write the variables and methods inside the class. Thus, class is binding them together. So class is an example for encapsulation.

The variables and methods of a class are called 'members' of the class. All the members of a class are by default available outside the class. That means they are *public* by default. Public means available to other programs and classes. Python follows *Uniform Access Principle* that says that in OOPS, all the members of the class whether they are variables or methods should be accessible in a uniform manner. So, Python variables and methods are available outside alike. That means both are *public* by default. Usually, in C++ and Java languages, the variables are kept *private*, that means they are not available outside the class and the methods are kept *public* meaning that they are available to other programs. But in Python, both the variables and methods are *public* by default.

Encapsulation isolates the members of a class from the members of another class. The reason is when objects are created, each object shares different memory and hence there will not be any overwriting of data. This gives an advantage to the programmer to use same names for the members of two different classes. For example, a programmer can declare and use the variables like 'id', 'name', and 'address' in different classes like Employee, Customer, or Student classes.

Encapsulation in Python

Encapsulation is nothing but writing attributes (variables) and methods inside a class. The methods process the data available in the variables. Hence data and code are bundled up together in the class. For example, we can write a Student class with 'id' and 'name as attributes along with the display() method that displays this data. This Student class becomes an example for encapsulation.

```
# a class is an example for encapsulation
class Student:
    # to declare and initialize the variables
    def __init__(self):
        self.id = 10
        self.name = 'Raju'

    # display students details
    def display(self):
        print(self.id)
        print(self.name)
```

Observe the first method: def __init__(self). This is called a special function since its name is starting and ending with two underscores. If a variable or method name starts and ends with two underscores, they are built-in variables or methods which are defined for a specific purpose. The programmer should not create any variable or method like them. It means we should not create variables or methods with two underscores before and after their names.

The purpose of the special method def __init__(self) is to declare and initialize the instance variables of a class. Instance variables are the variables whose copy is available in the object (or instance). The first parameter for this method is 'self' that represents the object (or instance) of the present class. So, self.id refers to the variable in the object. In the Student class, we have written another method by the name display() that displays the instance variables.

Abstraction

There may be a lot of data, a class contains and the user does not need the entire data. The user requires only some part of the available data. In this case, we can hide the unnecessary data from the user and expose only that data that is of interest to the user. This is called abstraction.

A good example for abstraction is a car. Any car will have some parts like engine, radiator, battery, mechanical and electrical equipment etc. The user of the car (driver) should know how to drive the car and does not require any knowledge of these parts. For example driver is never bothered about how the engine is designed and the internal parts of the engine. This is why the car manufacturers hide these parts from the driver in a separate panel, generally at the front of the car.

The advantage of abstraction is that every user will get his own view of the data according to his requirements and will not get confused with unnecessary data. A bank clerk should see the customer details like account number, name and balance amount in the account. He should not be entitled to see the sensitive data like the staff salaries, profit or loss of the bank, interest amount paid by the bank, loans amount to be recovered, etc. Hence, such sensitive data can be abstracted from the clerk's view. The bank manager may, however, require the sensitive data and so it will be provided to the manager.

Abstraction in Python

In languages like Java, we have keywords like private, protected and public to implement various levels of abstraction. These keywords are called *access specifiers*. In Python, such words are not available. Everything written in the class will come under public. That means everything written in the class is available outside the class to other people. Suppose, we do not want to make a variable available outside the class or to other members inside the class, we can write the variable with two double scores before it as: __var. This is like a private variable in Python. In the following example, 'y' is a private variable since it is written as: __y.

```
class Myclass:
    # this is constructor.
    def __init__(self):
        self.__y = 3    # this is private variable
```

Now, it is not possible to access the variable from within the class or out of the class as:

```
m = Myclass()
print(m.y)    # error
```

The preceding print() statement displays error message as: AttributeError: 'Myclass' object has no attribute 'y'. Even though, we cannot access the private variable in this way, it is possible to access it in the format: instancename._Classname__var. That means we are using Classname differently to access the private variable. This is called *name mangling*. In name mangling, we have to use one underscore before the classname and two underscores after the classname. Like this, using the names differently to access the private variables is called name mangling. For example, to display a private variable 'y' value, we can write:

```
print(m._Myclass__y)    # display private variable y
```

The same statement can be written inside the method as: print(self._Myclass__z). When we use single underscore before a variable as _var, then that variable or object will not be imported into other files. The following code represents the public and private variables and how to access them.

```
# understanding public and private variables
class Myclass:
    # this is constructor.
    def __init__(self):
        self.x = 1    # public var
        self.__y = 2  # private var
```

```
    # instance method to access variables
    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('Accessing variables through instance:')
print(m.x)   # x is available directly
print(m._Myclass__y)   # name mangling required
```

Output:

```
C:\>python oops.py
Accessing variables through method:
1
2
Accessing variables through instance:
1
2
```

We are planning to write Bank class with 'accno', 'name', 'balance' and 'loan' as variables. Since the clerk should not see the loan amount of the customer, we can write that variable with two underscores before the variable, as: ' __loan'. Then this variable is not available directly outside the class or inside the class to other methods.

In the Bank class, the first method is a special method with the name: __init__(self) is useful to declare variables and initialize them with some data. In the program, 'self' represents current class object. In this method, we are making loan variable as private by writing it as:

```
self. __loan = 1500000.00;
```

This variable is not available outside the class. It is not even available to other methods in the same class. Hence, it is abstracted completely from the user of the class. If the bank clerk calls the display_to_clerk(self) method, he will be able to see account number, name and balance amount only. He cannot see loan amount of the customer. That means some part of the data is hidden from the clerk. See the example code:

```
# accessing some part of data
class Bank :
    def __init__(self):
        self.accno = 10
        self.name = 'Srinu'
        self.balance = 5000.00
        self. __loan = 1500000.00

    def display_to_clerk(self):
        print(self.accno)
        print(self.name)
        print(self.balance)
```

In the preceding class, in spite of several data items, the display_to_clerk() method is able to access and display only the 'accno', 'name' and 'balance' values. It cannot access loan of the customer. This means the loan data is hidden from the view of the bank clerk. This is called abstraction. Suppose, we try to display the loan amount in the display_to_clerk() method, by writing:

```
print(self.loan)
```

This raises an error saying 'loan' is not an attribute of Bank class.

Inheritance

Creating new classes from existing classes, so that the new classes will acquire all the features of the existing classes is called Inheritance. A good example for Inheritance in nature is parents producing the children and children inheriting the qualities of the parents.

Let's take a class A with some members i.e., variables and methods. If we feel another class B wants almost same members, then we can derive or create class B from A as:

```
class B(A):
```

Now, all the features of A are available to B. If an object to B is created, it contains all the members of class A and also its own members. Thus, the programmer can access and use all the members of both the classes A and B. Thus, class B becomes more useful. This is called inheritance. The original class (A) is called the base class or super class and the derived class (B) is called the sub class or derived class.

There are three advantages of inheritance. First, we can create more useful classes needed by the application (software). Next, the process of creating the new classes is very easy, since they are built upon already existing classes. The last, but very important advantage is managing the code becomes easy, since the programmer creates several classes in a hierarchical manner, and segregates the code into several modules.

An Example for Inheritance in Python

Here, we take a class A with two variables 'a' and 'b' and a method, method1(). Since all these members are needed by another class B, we extend class B from A. We want some additional members in B, for example a variable 'c' and a method, method2(). So, these are written in B. Now remember, class B can use all the members of both A and B. This means the variables 'a','b','c 'and also the methods method1() and method2() are available to class B. That means all the members of A are inherited by B.

```
# Creating class B from class A
class A :
    a = 1
    b = 2
```

```
    def method1(cls):
        print(cls.a)
        print(cls.b)

class B(A):
    c = 3
    def method2(cls):
        print(cls.c)
```

By creating an object to B, we can access all the members of both the classes A and B.

Polymorphism

The word 'Polymorphism' came from two Greek words 'poly' meaning 'many' and 'morphos' meaning 'forms'. Thus, polymorphism represents the ability to assume several different forms. In programming, if an object or method is exhibiting different behavior in different contexts, it is called polymorphic nature.

Polymorphism provides flexibility in writing programs in such a way that the programmer uses same method call to perform different operations depending on the requirement.

Example Code for Polymorphism in Python:

When a function can perform different tasks, we can say that it is exhibiting polymorphism. A simple example is to write a function as:

```
def add(a, b):
    print(a+b)
```

Since in Python, there the variables are not declared explicitly, we are passing two variables 'a', and 'b' to add() function where they are added. While calling this function, if we pass two integers like 5 and 15, then this function displays 15. If we pass two strings, then the same function concatenates or joins those strings. That means the same function is adding two integers or concatenating two strings. Since the function is performing two different tasks, it is said to exhibit polymorphism. Now, consider the following example:

```
# a function that exhibits polymorphism
def add(a, b):
    print(a+b)

# call add() and pass two integers
add(5, 10)  # displays 15

# call add() and pass two strings
add("Core", "Python")  # displays CorePython
```

The programming languages which follow all the five features of OOPS are called object oriented programming languages. For example, C++, Java and Python will come into this category.

Points to Remember

- Procedure oriented approach is the methodology where programming is done using procedures and functions. This is followed by languages like C, Pascal and FORTRAN.
- Object oriented approach is the methodology where programming is done using classes and objects. This is followed in the languages like C++, Java and Python.
- Python programmers can write programs using procedure oriented approach (like C) or object oriented approach (like Java) depending on their requirements.
- An object is anything that really exists in the world and can be distinguished from others.
- Every object has some behavior that is characterized by attributes and actions. Attributes are represented by variables and actions are performed by methods. So an object contains variables and methods.
- A function written inside a class is called method.
- A class is a model or blueprint for creating objects. A class also contains variables and methods.
- Objects are created from a class.
- An object does not exist without a class; however, a class can exist without any object.
- Encapsulation is a mechanism where the data (variables) and the code (methods) that act on the data will bind together.
- Class is an example for encapsulation since it contains data and code.
- Hiding unnecessary data and code from the user is called abstraction.
- To hide variables or methods, we should declare them as private members. This is done by writing two underscores before the names of the variable or method.
- Private members can be accessed using name mangling where the class name is used with single underscore before it and two underscores after it in the form of: instancename. _Classname__variable or instancename. _Classname__method().
- Creating new classes from existing classes, so that new classes will acquire all the features of the existing classes is called Inheritance.

- In inheritance, the already existing class is called base class or super class. The newly created class is called sub class or derived class.
- Polymorphism represents the ability of an object or method to assume several different forms.
- The programming languages which follow all the five features of OOPS namely, classes and objects, encapsulation, abstraction, inheritance and polymorphism are called object oriented programming languages. For example, C++, Java and Python will come into this category.

Unit - 3

CHAPTER

13

CLASSES AND OBJECTS

We know that a class is a model or plan to create objects. This means, we write a class with the attributes and actions of objects. Attributes are represented by variables and actions are performed by methods. So, a class contains variable and methods. The same variables and methods are also available in the objects because they are created from the class. These variables are also called 'instance variables' because they are created inside the instance (i.e. object).

Please remember the difference between a function and a method. A function written inside a class is called a method. Generally, a method is called using one of the following two ways:

- classname.methodname()
- instancename.methodname()

The general format of a class is given as follows:

```
Class Classname(object):
    """ docstring describing the class """
    attributes
    def __init__(self):
    def method1():
    def method2():
```

Creating a Class

A class is created with the keyword *class* and then writing the Classname. After the Classname, 'object' is written inside the Classname. This 'object' represents the base class name from where all classes in Python are derived. Even our own classes are also derived from 'object' class. Hence, we should

mention 'object' in the parentheses. Please note that writing 'object' is not compulsory since it is implied.

The docstring is a string which is written using triple double quotes or triple single quotes that gives the complete description about the class and its usage. The docstring is used to create documentation file and hence it is optional. 'attributes' are nothing but variables that contains data. __init__(self) is a special method to initialize the variables. method1() and method2(), etc. are methods that are intended to process variables.

If we take 'Student' class, we can write code in the class that specifies the attributes and actions performed by any student. For example, a student has attributes like name, age, marks, etc. These attributes should be written inside the Student class as variables. Similarly, a student can perform actions like talking, writing, reading, etc. These actions should be represented by methods in the Student class. So, the class Student contains these attributes and actions, as shown here:

```
class Student:      # another way is: class Student(object):
    # the below block defines attributes
    def __init__(self):
        self.name = 'Vishnu'
        self.age = 20
        self.marks = 900

    # the below block defines a method
    def talk(self):
        print('Hi, I am ', self.name)
        print('My age is', self.age)
        print('My marks are', self.marks)
```

Observe that the keyword *class* is used to declare a class. After this, we should write the class name. So, 'Student' is our class name. Generally, a class name should start with a capital letter, hence 'S' is capital in 'Student'. In the class, we write attributes and methods. Since in Python, we cannot declare variables, we have written the variables inside a special method, i.e. __init__(). This method is useful to initialize the variables. Hence, the name 'init'. The method name has two underscores before and after. This indicates that this method is internally defined and we cannot call this method explicitly. Observe the parameter 'self' written after the method name in the parentheses. 'self' is a variable that refers to current class instance. When we create an instance for the Student class, a separate memory block is allocated on the heap and that memory location is by default stored in 'self'. The instance contains the variables 'name', 'age', 'marks' which are called *instance variables*. To refer to instance variables, we can use the dot operator notation along with self as: 'self.name', 'self.age' and 'self.marks'.

See the method talk(). This method also takes the 'self' variable as parameter. This method displays the values of the variables by referring them using 'self'.

The methods that act on instances (or objects) of a class are called *instance methods*. Instance methods use 'self' as the first parameter that refers to the location of the instance in the memory. Since instance methods know the location of instance, they can act on the instance variables. In the previous code, the two methods __init__(self) and talk(self) are called instance methods.

In the Student class, a student is talking to us through talk() method. He is introducing himself to us, as shown here:

```
Hi, I am Vishnu
My age is 20
My marks are 900
```

This is what the talk() method displays. Writing a class like this is not sufficient. It should be used. To use a class, we should create an instance (or object) to the class. Instance creation represents allotting memory necessary to store the actual data of the variables, i.e., Vishnu, 20 and 900. To create an instance, the following syntax is used:

```
instancename  = Classname()
```

So, to create an instance (or object) to the Student class, we can write as:

```
s1 = Student()
```

Here, 's1' is nothing but the instance name. When we create an instance like this, the following steps will take place internally:

1. First of all, a block of memory is allocated on heap. How much memory is to be allocated is decided from the attributes and methods available in the Student class.
2. After allocating the memory block, the special method by the name '__init__(self)' is called internally. This method stores the initial data into the variables. Since this method is useful to construct the instance, it is called 'constructor'.
3. Finally, the allocated memory location address of the instance is returned into 's1' variable. To see this memory location in decimal number format, we can use id() function as id(s1).

Now, 's1' refers to the instance of the Student class. Hence any variables or methods in the instance can be referenced by 's1' using dot operator as:

```
s1.name   # this refers to data in name variable, i.e. Vishnu
s1.age      # this refers to data in age variable, i.e. 20
s1.marks  # this refers to data in marks variable, i.e. 900
s1.talk()  # this calls the talk() method
```

The dot operator takes the instance name at its left and the member of the instance at the right hand side. Figure 13.1 shows how 's1' instance of Student class is created in memory:

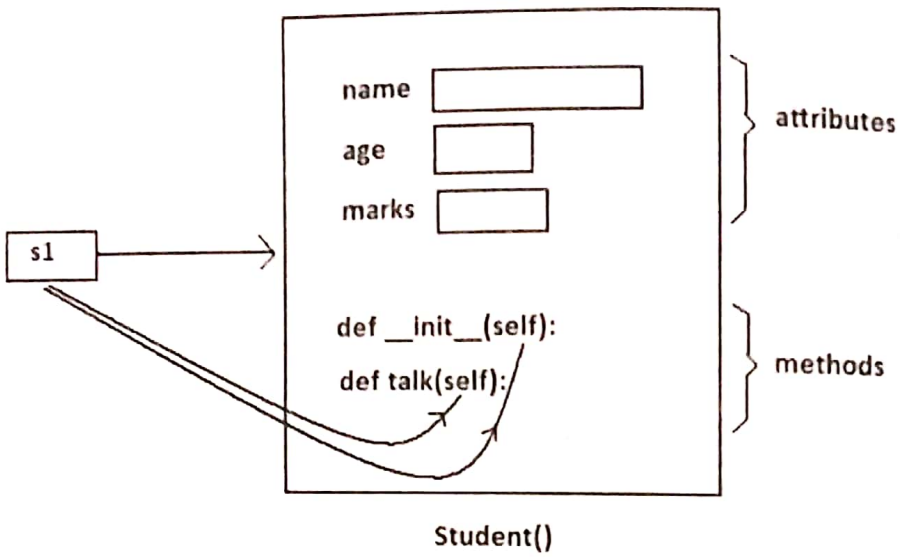

Figure 13.1: Student class instance in memory

Program

Program 1: A Python program to define Student class and create an object to it. Also, we will call the method and display the student's details.

```
# instance variables and instance method
class Student:
    # this is a special method called constructor.
    def __init__(self):
        self.name = 'Vishnu'
        self.age = 20
        self.marks = 900

# this is an instance method.
def talk(self):
    print('Hi, I am', self.name)
    print('My age is', self.age)
    print('My marks are', self.marks)

# create an instance to Student class.
s1 = Student()

# call the method using the instance.
s1.talk()
```

Output:

```
C:\>python c1.py
Hi, I am Vishnu
My age is 20
My marks are 900
```

In Program 1, we used the 'self' variable to refer to the instance of the same class. Also, we used a special method '__init__(self)' that initializes the variables of the instance. Let's have more clarity on these two concepts.

The Self Variable

'self' is a default variable that contains the memory address of the instance of the current class. So, we can use 'self' to refer to all the instance variables and instance methods.

When an instance to the class is created, the instance name contains the memory location of the instance. This memory location is internally passed to 'self'. For example, we create an instance to Student class as:

```
s1 = Student()
```

Here, 's1' contains the memory address of the instance. This memory address is internally and by default passed to 'self' variable. Since 'self' knows the memory address of the instance, it can refer to all the members of the instance. We use 'self' in two ways:

- The 'self' variable is used as first parameter in the constructor as:

```
def __init__(self):
```

  In this case, 'self' can be used to refer to the instance variables inside the constructor.

- 'self' can be used as first parameter in the instance methods as:

```
def talk(self):
```

  Here, talk() is instance method as it acts on the instance variables. If this method wants to act on the instance variables, it should know the memory location of the instance variables. That memory location is by default available to the talk() method through 'self'.

Constructor

A constructor is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be 'self' variable that contains the memory address of the instance. For example,

```
def __init__(self):
    self.name = 'Vishnu'
    self.marks = 900
```

Here, the constructor has only one parameter, i.e. 'self'. Using 'self.name' and 'self.marks', we can access the instance variables of the class. A constructor is called at the time of creating an instance. So, the above constructor will be called when we create an instance as:

```
s1 = Student()
```

Here, 's1' is the name of the instance. Observe the empty parentheses after the class name 'Student'. These empty parentheses represent that we are not passing any values to the constructor. Suppose, we want to pass some values to the constructor, then we have to pass them in the parentheses after the class name. Let's take another example. We can write a constructor with some parameters in addition to 'self' as:

```
def __init__(self, n = '', m=0):
    self.name = n
    self.marks = m
```

Here, the formal arguments are 'n' and 'm' whose default values are given as '' (None) and 0 (zero). Hence, if we do not pass any values to constructor at the time of creating an instance, the default values of these formal arguments are stored into name and marks variables. For example,

```
s1 = Student()
```

Since we are not passing any values to the instance, None and zero are stored into name and marks. Suppose, we create an instance as:

```
s1 = Student('Lakshmi Roy', 880)
```

In this case, we are passing two actual arguments: 'Lakshmi Roy' and 880 to the Student instance. Hence these values are sent to the arguments 'n' and 'm' and from there stored into name and marks variables. We can understand this concept from Program 2.

Program

Program 2: A Python program to create Student class with a constructor having more than one parameter.

```
# instance vars and instance method - v.20
class Student:
    # this is constructor.
    def __init__(self, n = '', m=0):
        self.name = n
        self.marks = m

    # this is an instance method.
    def display(self):
        print('Hi', self.name)
        print('Your marks', self.marks)

# constructor is called without any arguments
s = Student()
s.display()
print('------------------')

# constructor is called with 2 arguments
s1 = Student('Lakshmi Roy', 880)
s1.display()
print('------------------')
```

Output:

```
C:\>python c1.py
Hi
Your marks 0
```

```
----------------
Hi Lakshmi Roy
Your marks 880
----------------
```

We should understand that a constructor does not create an instance. The duty of the constructor is to initialize or store the beginning values into the instance variables. A constructor is called only once at the time of creating an instance. Thus, if 3 instances are created for a class, the constructor will be called once per each instance, thus it is called 3 times.

Types of Variables

The variables which are written inside a class are of 2 types:

- Instance variables
- Class variables or Static variables

Instance variables are the variables whose separate copy is created in every instance (or object). For example, if 'x' is an instance variable and if we create 3 instances, there will be 3 copies of 'x' in these 3 instances. When we modify the copy of 'x' in any instance, it will not modify the other two copies. Consider Program 3.

Program

Program 3: A Python program to understand instance variables.

```
# instance vars example
class Sample:
    # this is a constructor.
    def __init__(self):
        self.x = 10

    # this is an instance method.
    def modify(self):
        self.x+=1

# create 2 instances
s1 = Sample()
s2 = Sample()
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)

# modify x in s1
s1.modify()
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
```

Output:

```
C:\>python c1.py
x in s1= 10
x in s2= 10
x in s1= 11
x in s2= 10
```

Instance variables are defined and initialized using a constructor with 'self' parameter. Also, to access instance variables, we need instance methods with 'self' as first parameter. It is possible that the instance methods may have other parameters in addition to the 'self' parameter. To access the instance variables, we can use self.variable as shown in Program 3. It is also possible to access the instance variables from outside the class, as: instancename.variable, e.g. s1.x.

Unlike instance variables, class variables are the variables whose single copy is available to all the instances of the class. If we modify the copy of class variable in an instance, it will modify all the copies in the other instances. For example, if 'x' is a class variable and if we create 3 instances, the same copy of 'x' is passed to these 3 instances. When we modify the copy of 'x' in any instance using a class method, the modified copy is sent to the other two instances. This can be easily grasped from Program 4. Class variables are also called static variables.

Program

Program 4: A Python program to understand class variables or static variables.

```
# class vars or static vars example
class Sample:
    # this is a class var
    x = 10

    # this is a class method.
    @classmethod
    def modify(cls):
        cls.x+=1

# create 2 instances
s1 = Sample()
s2 = Sample()
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)

# modify x in s1
s1.modify()
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
```

Output:

```
C:\>python cl.py
x in s1= 10
x in s2= 10
x in s1= 11
x in s2= 11
```

Observe Program 4. The class variable 'x' is defined in the class and initialized with value 10. A method by the name 'modify' is used to modify the value of 'x'. This method is called 'class method' since it is acting on the class variable. To mark this method as class method, we should use built-in decorator statement @classmethod. For example,

```
@classmethod   # this is a decorator
    def modify(cls):   # cls must be the first parameter
        cls.x+=1   # cls.x refers to class variable x
```

A class method contains first parameter by default as 'cls' with which we can access the class variables. For example, to refer to the class variable 'x', we can use 'cls.x'. We can also write other parameters in the class method in addition to the 'cls' parameter. The point is that the class variables are defined directly in the class. To access class variables, we need class methods with 'cls' as first parameter. We can access the class variables using the class methods as: cls.variable. If we want to access the class variables from outside the class, we can use: classname.variable, e.g. Sample.x.

Namespaces

A *namespace* represents a memory block where names are mapped (or linked) to objects. Suppose we write:

```
n = 10
```

Here, 'n' is the name given to the integer object 10. Please recollect that numbers, strings, lists etc. are all considered as objects in Python. The name 'n' is linked to 10 in the namespace. A class maintains its own namespace, called 'class namespace'. In the class namespace, the names are mapped to class variables. Similarly, every instance will have its own name space, called 'instance namespace'. In the instance namespace, the names are mapped to instance variables. In the following code, 'n' is a class variable in the Student class. So, in the class namespace, the name 'n' is mapped or linked to 10 as shown Figure 13.2. Since it is a class variable, we can access it in the class namespace, using classname.variable, as: Student.n which gives 10.

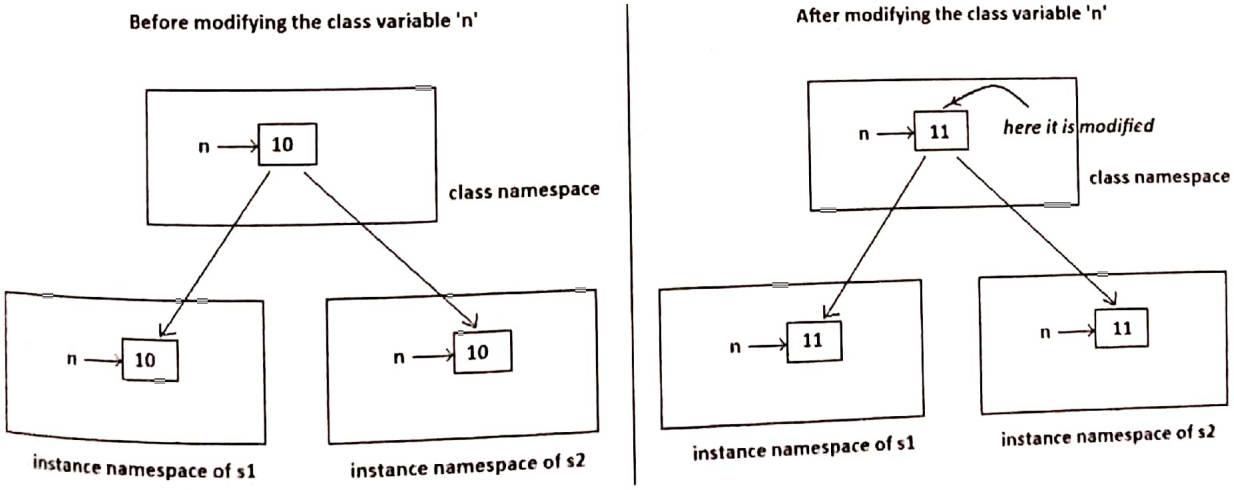

Figure 13.2: Modifying the class variable in the class namespace

```
# understanding class namespace
class Student:
    # this is a class var
    n=10

# access class var in the class namespace
print(Student.n)    # displays 10
Student.n+=1        # modify jt in class namespace
print(Student.n)    # displays 11
```

We know that a single copy of class variable is shared by all the instances. So, if the class variable is modified in the class namespace, since same copy of the variable is modified, the modified copy is available to all the instances. This is shown in Figure 13.2.

```
# modified class var is seen in all instances
s1 = Student()    # create s1 instance
print(s1.n)       # displays 11
s2 = Student()    # create s2 instance
print(s2.n)       # displays 11
```

What happens when the class variable is modified in the instance namespace? Since every instance will have its own namespace, if the class variable is modified in one instance namespace, it will not affect the variables in the other instance namespaces. This is shown in Figure 13.3. To access the class variable at the instance level, we have to create instance first and then refer to the variable as instancename.variable.

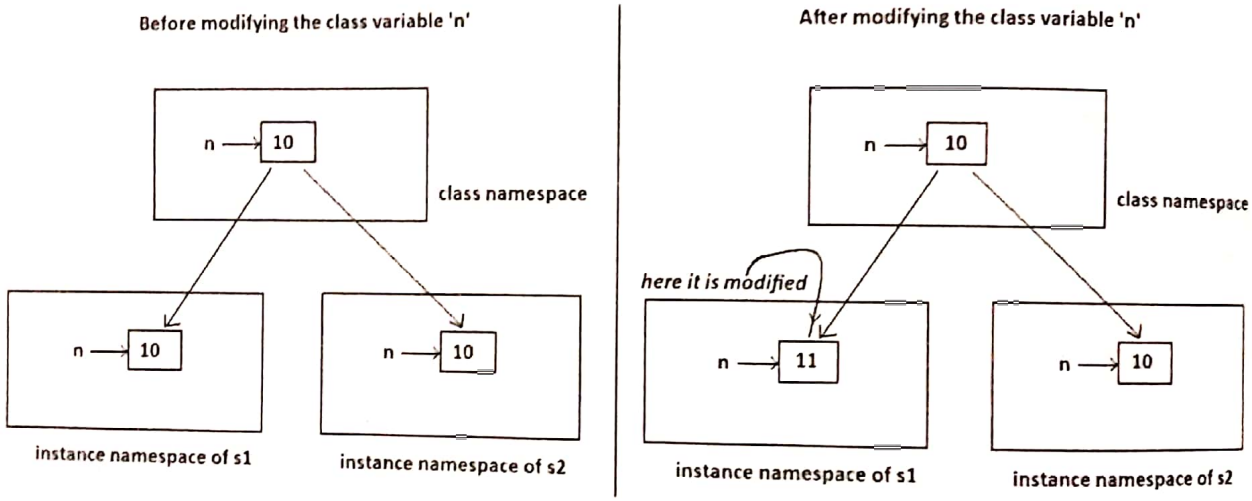

Figure 13.3: Modifying the class variable in the instance namespace

```
# understanding instance namespace
class Student:
    # this is a class var
    n=10

# access class var in the s1 instance namespace
s1 = Student()
print(s1.n)    # displays 10
s1.n+=1    # modify it in s1 instance namespace
print(s1.n)    # displays 11
```

As per the above code, we created an instance 's1' and modified the class variable 'n' in that instance. So, the modified value of 'n' can be seen only in that instance. When we create other instances like 's2', there will be still the original value of 'n' available. See the code below:

```
# modified class var is not seen in other instances
s2 = Student()    # this is another instance
print(s2.n)    # displays 10, not 11
```

Types of Methods

By this time, we got some knowledge about the methods written in a class. The purpose of a method is to process the variables provided in the class or in the method. We already know that the variables declared in the class are called class variables (or static variables) and the variables declared in the constructor are called instance variables. We can classify the methods in the following 3 types:

- Instance methods
  - (a) Accessor methods
  - (b) Mutator methods
- Class methods
- Static methods

Instance Methods

Instance methods are the methods which act upon the instance variables of the class. Instance methods are bound to instances (or objects) and hence called as: instancename.method(). Since instance variables are available in the instance, instance methods need to know the memory address of the instance. This is provided through 'self' variable by default as first parameter for the instance method. While calling the instance methods, we need not pass any value to the 'self' variable.

Program 5 is an extension to our previous Student class. In this program, we are creating a Student class with a constructor that defines 'name' and 'marks' as instance variables. An instance method display() will display the values of these variables. We added another instance methods by the name calculate() that calculates the grades of the student depending on the 'marks'.

Program

Program 5: A Python program using a student class with instance methods to process the data of several students.

```
# instance methods to process data of the objects
class Student:
    # this is a constructor.
    def __init__(self, n = '', m=0):
        self.name = n
        self.marks = m
    # this is an instance method.
    def display(self):
        print('Hi', self.name)
        print('Your marks', self.marks)

    # to calculate grades based on marks.
    def calculate(self):
        if(self.marks>=600):
            print('You got first grade')
```

```
            elif(self.marks>=500):
                print('You got second grade')
            elif(self.marks>=350):
                print('You got third grade')
            else:
                print('You are failed')

    # create instances with some data from keyboard
    n = int(input('How many students? '))

    i=0
    while(i<n):
        name = input('Enter name: ')
        marks = int(input('Enter marks: '))

        # create Student class instance and store data
        s = Student(name, marks)
        s.display()
        s.calculate()
        i+=1
        print('----------------------')
```

Output:

```
C:\>python c1.py
How many students? 3
Enter name: Vishnu Vardhan
Enter marks: 800
Hi Vishnu Vardhan
Your marks 800
You got first grade
----------------------
Enter name: Tilak Prabhu
Enter marks: 360
Hi Tilak Prabhu
Your marks 360
You got third grade
----------------------
Enter name: Gunasheela
Enter marks: 550
Hi Gunasheela
Your marks 550
You got second grade
----------------------
```

Instance methods are of two types: accessor methods and mutator methods. Accessor methods simply access or read data of the variables. They do not modify the data in the variables. Accessor methods are generally written in the form of getXXX() and hence they are also called *getter* methods. For example,

```
def getName(self):
    return self.name
```

Here, getName() is an accessor method since it is reading and returning the value of 'name' instance variable. It is not modifying the value of the name variable. On the other hand, mutator methods are the methods which not only read the data but also modify them. They are written in the form of setXXX() and hence they are also called *setter* methods. For example,

```
    def setName(self, name):
        self.name = name
```

Here, setName() is a mutator method since it is modifying the value of 'name' variable by storing new name. In the method body, 'self.name' represents the instance variable 'name' and the right hand side 'name' indicates the parameter that receives the new value from outside. In Program 6, we are redeveloping the Student class using accessor and mutator methods.

Program

Program 6: A Python program to store data into instances using mutator methods and to retrieve data from the instances using accessor methods.

```
# accessor and mutator methods
class Student:

    # mutator method
    def setName(self, name):
        self.name = name

    # accessor method
    def getName(self):
        return self.name

    # mutator method
    def setMarks(self, marks):
        self.marks = marks

    # accessor method
    def getMarks(self):
        return self.marks

# create instances with some data from keyboard
n = int(input('How many students? '))

i=0
while(i<n):
    # create Student class instance
    s = Student()
    name = input('Enter name: ')
    s.setName(name)
    marks = int(input('Enter marks: '))
    s.setMarks(marks)

    # retrieve data from Student class instance
    print('Hi', s.getName())
    print('Your marks', s.getMarks())

    i+=1
    print('--------------------')
```

Output:

```
C:\>python c1.py
How many students? 2
Enter name: Vinay Krishna
Enter marks: 890
Hi Vinay Krishna
```

```
Your marks 890
------------------
Enter name: Vimala Rao
Enter marks: 750
Hi Vimala Rao
Your marks 750
------------------
```

Since mutator methods define the instance variables and store data, we need not write the constructor in the class to initialize the instance variables. This is the reason we did not use constructor in Student class in Program 6.

Class Methods

These methods act on class level. Class methods are the methods which act on the class variables or static variables. These methods are written using @classmethod decorator above them. By default, the first parameter for class methods is 'cls' which refers to the class itself. For example, 'cls.var' is the format to refer to the class variable. These methods are generally called using the classname.method(). The processing which is commonly needed by all the instances of the class is handled by the class methods. In Program 7, we are going to develop Bird class. All birds in the Nature have only 2 wings. So, we take 'wings' as a class variable. Now a copy of this class variable is available to all the instances of Bird class. The class method fly() can be called as Bird.fly().

Program

Program 7: A Python program to use class method to handle the common feature of all the instances of Bird class.

```
# understanding class methods
class Bird:
    # this is a class var
    wings = 2

    # this is a class method
    @classmethod
    def fly(cls, name):
        print('{} flies with {} wings'.format(name, cls.wings))

# display information for 2 birds
Bird.fly('Sparrow')
Bird.fly('Pigeon')
```

Output:

```
C:\>python cl.py
Sparrow flies with 2 wings
Pigeon flies with 2 wings
```

Static Methods

We need static methods when the processing is at the class level but we need not involve the class or instances. Static methods are used when some processing is related to the class but does not need the class or its instances to perform any work. For example,

setting environmental variables, counting the number of instances of the class or changing an attribute in another class, etc. are the tasks related to a class. Such tasks are handled by static methods. Also, static methods can be used to accept some values, process them and return the result. In this case the involvement of neither the class nor the objects is needed. Static methods are written with a decorator @staticmethod above them. Static methods are called in the form of classname.method(). In Program 8, we are creating a static method noObjects() that counts the number of objects or instances created to Myclass. In Myclass, we have written a constructor that increments the class variable 'n' every time an instance is created. This incremented value of 'n' is displayed by the noObjects() method.

Program

Program 8: A Python program to create a static method that counts the number of instances created for a class.

```
# understanding static methods
class Myclass:
    # this is class var or static var
    n=0

    # constructor that increments n when an instance is created
    def __init__(self):
        Myclass.n = Myclass.n+1

    # this is a static method to display the no. of instances
    @staticmethod
    def noObjects():
        print('No. of instances created: ', Myclass.n)

# create 3 instances
obj1 = Myclass()
obj2 = Myclass()
obj3 = Myclass()
Myclass.noObjects()
```

Output:

```
C:\>python cl.py
No. of instances created: 3
```

In the next program, we accept a number from the keyboard and return the result of its square root value. Here, there is no need of class or object and hence we can write a static method to perform this task.

Program

Program 9: A Python program to create a static method that calculates the square root value of a given number.

```
# a static method to find square root value
import math
class Sample:
    @staticmethod
    def calculate(x):
```

```
        result = math.sqrt(x)
        return result

# accept a number from keyboard
num = float(input('Enter a number: '))

# call the static method and pass num
res = Sample.calculate(num)
print('The square root of {} is {:.2f}'.format(num, res))
```

Output:

```
C:\>python calc.py
Enter a number: 49
The square root of 49.0 is 7.00
```

In Program 10, we are creating a Bank class. An account in the bank is characterized by name of the customer and balance amount in the account. Hence a constructor is written that defines 'name' and 'balance' attributes. If balance is not given, then it is taken as 0.0. The deposit() method is useful to handle the deposits and the withdraw() method is useful to handle the withdrawals. Bank class can be used by creating an instance 'b' to it as:

```
b = Bank(name)
```

Since the constructor expects the name of the customer, we have to pass the name in the parentheses while creating the instance of the Bank class. In the while loop, we are displaying a one line menu as:

```
print('d -Deposit, w -Withdraw, e -Exit')
```

When the user choice is 'e' or 'E', we will terminate the program by calling the exit() method of 'sys' module.

Program

Program 10: A Python program to create a Bank class where deposits and withdrawals can be handled by using instance methods.

```
# A class to handle deposits and withdrawals in a bank
import sys
class Bank(object):
    """ Bank related transactions """

    # to initialize name and balance instance vars
    def __init__(self, name, balance=0.0):
        self.name = name
        self.balance = balance

    # to add deposit amount to balance
    def deposit(self, amount):
        self.balance += amount
        return self.balance

    # to deduct withdrawal amount from balance
    def withdraw(self, amount):
        if amount >self.balance:
            print('Balance amount is less, so no withdrawal.')
        else:
```

```
            self.balance -= amount
        return self.balance

# using the Bank class
# create an account with the given name and balance 0.00
name = input('Enter name: ')
b = Bank(name)    # this is instance of Bank class

# repeat continuously till choice is 'e' or 'E'.
while(True):
    print('d -Deposit, w -Withdraw, e -Exit')
    choice = input('Your choice: ')
    if choice == 'e' or choice == 'E':
        sys.exit()
    # amount for deposit or withdraw
    amt = float(input('Enter amount: '))

    # do the transaction
    if choice == 'd' or choice == 'D':
        print('Balance after deposit: ', b.deposit(amt))
    elif choice == 'w' or choice == 'W':
        print('Balance after withdrawal: ', b.withdraw(amt))
```

Output:

```
C:\>python cl.py
Enter name: Madhuri
d -Deposit, w -Withdraw, e -Exit
Your choice: d
Enter amount: 10000
Balance after deposit: 10000.0
d -Deposit, w -Withdraw, e -Exit
Your choice: w
Enter amount: 3500
Balance after withdrawal: 6500.0
d -Deposit, w -Withdraw, e -Exit
Your choice: e
```

Passing Members of One Class to Another Class

It is possible to pass the members (i.e. attributes and methods) of a class to another class. Let's take an Emp class with a constructor that defines attributes 'id', 'name', and 'salary'. This class has an instance method display() to display these values. If we create an object (or instance) of Emp class, it contains a copy of all the attributes and methods. To pass all these members of Emp class to another class, we should pass Emp class instance to the other class. For example, let's create an instance of Emp class as:

```
e = Emp()
```

Then pass this instance 'e' to a method of other class, as:

```
Myclass.mymethod(e)
```

Here, Myclass is the other class and mymethod() is a static method that belongs to Myclass. In Myclass, the method mymethod() will be declared as a static method as it acts neither on the class variables nor instance variables of Myclass. The purpose of

mymethod() is to change the attribute of Emp class. For example, mymethod() may increment the employee salary by 1000 rupees as shown below:

```
def mymethod(e):
    # increment salary in e by 1000
    e.salary+=1000;   # modify attribute of Emp class
    e.display()   # call the method of Emp class
```

So, the point is this: by passing the instance of a class, we are passing all the attributes and methods to another class. In the other class, it is possible to utilize them as needed. In our example, Myclass method, i.e. mymethod() is utilizing the salary attribute and display() methods of Emp class.

Program

Program 11: A Python program to create Emp class and make all the members of the Emp class available to another class, i.e. Myclass.

```
# this class contains employee details
class Emp:
    # this is a constructor.
    def __init__(self, id, name, salary):
        self.id = id
        self.name = name
        self.salary = salary

    # this is an instance method.
    def display(self):
        print('Id=', self.id)
        print('Name=', self.name)
        print('Salary= ', self.salary)

# this class displays employee details
class Myclass:
    # method to receive Emp class instance
    # and display employee details
    @staticmethod
    def mymethod(e):
        # increment salary of e by 1000
        e.salary+=1000;
        e.display()

# create Emp class instance e
e = Emp(10, 'Raj kumar', 15000.75)
# call static method of Myclass and pass e
Myclass.mymethod(e)
```

Output:

```
C:\>python cl.py
Id= 10
Name= Raj kumar
Salary= 16000.75
```

Let's understand that static methods are used when the class variables or instance variables are not disturbed. We have to use a static method when we want to pass some values from outside and perform some calculation in the method. Here, we are not

touching the class variable or instance variables. Program 12 shows a static method that calculates the value of a number raised to a power.

Program

Program 12: A Python program to calculate power value of a number with the help of a static method.

```
# another example for static method
class Myclass:
    # method to calculate x to the power of n
    @staticmethod
    def mymethod(x, n):
        result = x**n
        print('{} to the power of {} is {}'.format(x, n, result))

# call the static method
Myclass.mymethod(5, 3)
Myclass.mymethod(5, 4)
```

Output:

```
C:\>python c1.py
5 to the power of 3 is 125
5 to the power of 4 Is 625
```

Inner Classes

Writing a class within another class is called creating an inner class or nested class. For example, if we write class B inside class A, then B is called inner class or nested class. Inner classes are useful when we want to sub group the data of a class. For example, let's take a person's data like name, age, date of birth etc. Here, name contains a single value like 'Charles', age contains a single value like '30' but the date of birth does not contain a single value. Rather, it contains three values like date, month and year. So, we need to take these three values as a sub group. Hence it is better to write date of birth as a separate class Dob inside the Person class. This Dob will contain instance variables dd, mm and yy which represent the date of birth details of the person.

Generally, the inner class object is created within the outer class. Let's take Person class as outer class and Dob as inner class. Dob class object is created in the constructor of the Person class as:

```
class Person:
    def __init__(self):
        self.name = 'Charles'
        self.db = self.Dob()   # this is Dob object
```

In the preceding code, 'db' represents the inner class object. When the outer class object is created, it contains a sub object that is inner class object. Hence, we can refer outer class and inner class members as:

```
p = Person()   # create outer class object
p.display()   # call outer class method
print(p.name)   # refer to outer class instance variable
```

```
x = p.db   # create inner class object
x.display()   # call inner class method
print(x.yy)   # refer to inner class instance variable
```

Program

Program 13: A Python program to create Dob class within Person class.

```
# inner class example
class Person:
    def __init__(self):
        self.name = 'Charles'
        self.db = self.Dob()

    def display(self):
        print('Name= ', self.name)

    # this is inner class
    class Dob:
        def __init__(self):
            self.dd = 10
            self.mm = 5
            self.yy = 1988
        def display(self):
            print('Dob= {}/{}/{}'.format(self.dd, self.mm, self.yy))

# creating Person class object
p = Person()
p.display()

# create inner class object
x = p.db
x.display()
```

Output:

```
C:\>python inner.py
Name= Charles
Dob= 10/5/1988
```

It is not compulsory to create inner class object inside outer class. That means, we need not write the following statement in the Person class constructor:

```
self.db = self.Dob()
```

If we do not write this, then there is no relation between the outer class object and inner class object. Then how to refer to inner class members is the question. In this case, we have to create outer class object and then using dot operator, we should mention the inner class object as:

```
x = Person().Dob()   # create inner class object
x.display()   # call inner class method
print(x.yy)   # refer to inner class instance variable
```

This concept is shown in Program 14.

Program 14: A Python program to create another version of Dob class within Person class.

```
# inner class example - v2.0
class Person:
    def __init__(self):
        self.name = 'Charles'

    def display(self):
        print('Name= ', self.name)

    # this is inner class
    class Dob:
        def __init__(self):
            self.dd = 10
            self.mm = 5
            self.yy = 1988

        def display(self):
            print('Dob= {}/{}/{}'.format(self.dd, self.mm, self.yy))

# creating Person class object
p = Person()
p.display()
# create Dob class object as sub object to Person class object
x = Person().Dob()
x.display()
print(x.yy)
```

Output:

```
C:\>python inner.py
Name= Charles
Dob= 10/5/1988
1988
```

Points to Remember

- A class is a model or plan to create objects. The objects of a class are also called instances.
- A class contains attributes which are nothing but variables.
- A class can contain methods which are useful to process variables.
- A method is a function that is written inside a class.
- The object class is the base class from where all other classes are derived.
- To use a class, generally we should create an instance. To create an instance of a class, we can write:

  instancename = Classname()
- A constructor is a special method that is useful to declare and initialize the instance variables. The general format of the constructor is:

  def __init__(self, parameters):

- A constructor is called only once at the time of creating an instance or object.
- 'self' is a variable that contains by default the memory address of the instance. We need not pass anything to this variable.
- The 'self' variable becomes the first parameter for constructor and instance methods.
- Instance variables are the variables whose separate copy is created in every instance (or object). Instance variables are referenced as instancename.var.
- Class variables are the variables whose single copy is available to all the instances of the class. Class variables are also called static variables. Class variables are referenced as classname.var.
- A namespace represents a memory block where names are mapped (or linked) to objects. A class will have class namespace and an instance will have its own namespace called instance namespace.
- The variables in the class namespace are referenced as classname.var. The variables in the instance namespace are referenced as instancename.var.
- Instance methods are the methods which act upon the instance variables of the class. Instance methods are bound to instances (or objects) and hence called as: instancename.method().
- Instance methods use 'self' as first default parameter.
- Instance methods are again classified as accessor methods and mutator methods. Accessor methods access or read the instance variables; whereas, mutator methods not only read the instance variables but also modify them.
- Generally, accessor methods are written in the form of getXXX() and hence they are also called getter methods. Mutator methods are written in the form of setXXX() and hence are called setter methods.
- Class methods are the methods which act on the class variables or static variables. These methods are written using @classmethod decorator above them. These methods are generally called using the classname.method().
- Class methods are written using 'cls' as their first parameter.
- Static methods are used when some processing is related to the class but does not need the class or its instances to perform any work. Static methods are written with a decorator @staticmethod above them. Static methods are called in the form of classname.method().
- It is possible to create inner classes within a class. To refer to the inner class members, we can create inner class object as:

innerobj = Outerclass().Innerclass()

innerobj = outerobj.innerobj

INHERITANCE AND POLYMORPHISM

CHAPTER 14

Software development is a team effort. Several programmers will work as a team to develop software. When a programmer develops a class, he will use its features by creating an instance to it. When another programmer wants to create another class which is similar to the class already created, then he need not create the class from the scratch. He can simply use the features of the existing class in creating his own class. Let's take an example to understand this.

A programmer in the software development is creating Teacher class with setter() and getter() methods as shown in Program 1. Then he saved this code in a file 'teacher.py'.

Program

Program 1: A Python program to create Teacher class and store it into teacher.py module.

```
# this is Teacher class. save this code in teacher.py file
class Teacher:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self, name):
        self.name = name

    def getname(self):
        return self.name

    def setaddress(self, address):
        self.address = address

    def getaddress(self):
        return self.address
```

```
    def setsalary(self, salary):
        self.salary = salary

    def getsalary(self):
        return self.salary
```

When the programmer wants to use this Teacher class that is available in teacher.py file, he can simply import this class into his program and use it as shown here:

Program

Program 2: A Python program to use the Teacher class.

```
# save this code as inh.py file
# using Teacher class
from teacher import Teacher
# create instance
t = Teacher()

# store data into the instance
t.setid(10)
t.setname('Prakash')
t.setaddress('HNO-10, Rajouri gardens, Delhi')
t.setsalary(25000.50)

# retrieve data from instance and display
print('id= ', t.getid())
print('name= ', t.getname())
print('address= ', t.getaddress())
print('salary= ', t.getsalary())
```

Output:

```
C:\>python inh.py
id= 10
name= Prakash
address= HNO-10, Rajouri gardens, Delhi
salary= 25000.5
```

So, the program is working well. There is no problem. Once the Teacher class is completed, the programmer stored teacher.py program in a central database that is available to all the members of the team. So, Teacher class is made available through the module teacher.py, as shown in Figure 14.1:

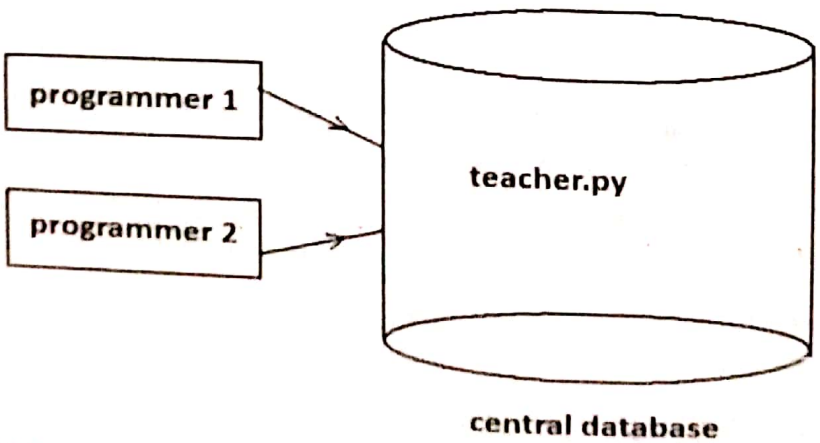

Figure 14.1: The teacher.py module is created and available in the project database

Now, another programmer in the same team wants to create a Student class. He is planning the Student class without considering the Teacher class as shown in Program 3.

Program 3: A Python program to create Student class and store it into student.py module.

```
# this is Student class -v1.0. save it as student.py
class Student:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self, name):
        self.name = name

    def getname(self):
        return self.name

    def setaddress(self, address):
        self.address = address

    def getaddress(self):
        return self.address

    def setmarks(self, marks):
        self.marks = marks

    def getmarks(self):
        return self.marks
```

Now, the second programmer who created this Student class and saved it as student.py can use it whenever he needs. Using the Student class is shown in Program 4.

Program 4: A Python program to use the Student class which is already available in student.py

```
# save this code as inh.py
# using Student class
from student import Student
# create instance
s = Student()

# store data into the instance
s.setid(100)
s.setname('Rakesh')
s.setaddress('HNO-22, Ameerpet, Hyderabad')
s.setmarks(970)
```

```
# retrieve data from instance and display
print('id= ', s.getid())
print('name= ', s.getname())
print('address= ', s.getaddress())
print('marks= ', s.getmarks())
```

Output:

```
C:\>python inh.py
id= 100
name= Rakesh
address= HNO-22, Ameerpet, Hyderabad
marks= 970
```

So far, so nice! If we compare the Teacher class and the Student classes, we can understand that 75% of the code is same in both the classes. That means most of the code being planned by the second programmer in his Student class is already available in the Teacher class. Then why doesn't he use it for his advantage? Our idea is this: instead of creating a new class altogether, he can reuse the code which is already available. This is shown in Program 5.

Program

Program 5:A Python program to create Student class by deriving it from the Teacher class.

```
# Student class - v2.0.save it as student.py
from teacher import Teacher
class Student(Teacher):
    def setmarks(self, marks):
        self.marks = marks

    def getmarks(self):
        return self.marks
```

The preceding code will be same as the first version of the Student class. Observe this code. In the first statement we are importing Teacher class from teacher module so that the Teacher class is now available to this program. Then we are creating Student class as:

```
class Student(Teacher):
```

This means the Student class is derived from Teacher class. Once we write like this, all the members of Teacher class are available to the Student class. Hence we can use them without rewriting them in the Student class. In addition, the following two methods are needed by the Student class but not available in the Teacher class:

```
def setmarks(self, marks):
def getmarks(self):
```

Hence, we wrote only these two methods in the Student class. Now, we can use the Student class as we did earlier. Creating the instance to the Student class and calling the methods as:

```
# create instance
s = Student()
# store data into the instance
s.setid(100)
s.setname('Rakesh')
s.setaddress('HNO-22, Ameerpet, Hyderabad')
s.setmarks(970)

# retrieve data from instance and display
print('id= ', s.getid())
print('name= ', s.getname())
print('address= ', s.getaddress())
print('marks= ', s.getmarks())
```

In other words, we can say that we have created Student class from the Teacher class. This is called inheritance. The original class, i.e. Teacher class is called base class or super class and the newly created class, i.e. the Student class is called the sub class or derived class. So, how can we define inheritance? Deriving new classes from the existing classes such that the new classes inherit all the members of the existing classes, is called inheritance. The syntax for inheritance is:

```
class Subclass(Baseclass):
```

The next question is why the base class members are automatically available to sub class? When an object to Student class is created, it contains a copy of Teacher class within it. This means there is a relation between the Teacher class and Student class objects. This is the reason Teacher class members are available to Student class. Note that we do not create Teacher class object, but still a copy of it is available to Student class object. Please see the object diagram of Student class in Figure 14.2. We can understand that all the members (i.e., variables and methods) of Teacher class as well as Student class are available in the Student class object.

Then, what is the advantage of inheritance? Please look at Student class version 1 and Student class version 2. Clearly, second version is smaller and easier to develop. By using inheritance, a programmer can develop the classes very easily. Hence programmer's productivity is increased. Productivity is a term that refers to the code developed by the programmer in a given span of time. If the programmer used inheritance, he will be able to develop more code in less time. So, his productivity is increased. This will increase the overall productivity of the organization, which means more profits for the organization and better growth for the programmer.

In inheritance, we always create only the sub class object. Generally, we do not create super class object. The reason is clear. Since all the members of the super class are available to sub class, when we create an object, we can access the members of both the super and sub classes. But if we create an object to super class, we can access only the super class members and not the sub class members.

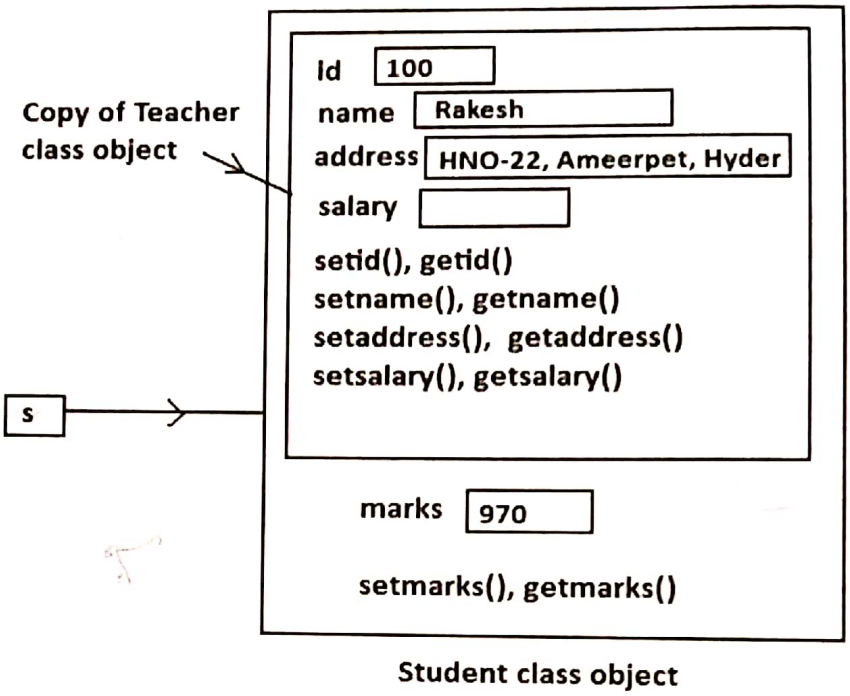

Figure 14.2: Student class object contains a copy of Teacher class object

Constructors in Inheritance

In the previous programs, we have inherited the Student class from the Teacher class. All the methods and the variables in those methods of the Teacher class (base class) are accessible to the Student class (sub class). Are the constructors of the base class accessible to the sub class or not – is the next question we will answer. In Program 6, we are taking a super class by the name 'Father' and derived a sub class 'Son' from it. The Father class has a constructor where a variable 'property' is declared and initialized with 800000.00. When Son is created from Father, this constructor is by default available to Son class. When we call the method of the super class using sub class object, it will display the value of the 'property' variable.

Program

Program 6: A Python program to access the base class constructor from sub class.

```
# base class constructor is available to sub class
class Father:
    def __init__(self):
        self.property = 800000.00

    def display_property(self):
        print('Father\'s property= ', self.property)

class Son(Father):
    pass    # we do not want to write anything in the sub class
```

```
# create sub class instance and display father's property
s = Son()
s.display_property()
```

Output:

```
C:\>python inh.py
Father's property= 800000.0
```

The conclusion is this: like the variables and methods, the constructors in the super class are also available to the sub class object by default.

Overriding Super Class Constructors and Methods

When the programmer writes a constructor in the sub class, the super class constructor is not available to the sub class. In this case, only the sub class constructor is accessible from the sub class object. That means the sub class constructor is replacing the super class constructor. This is called *constructor overriding*. Similarly in the sub class, if we write a method with exactly same name as that of super class method, it will override the super class method. This is called *method overriding*. Consider Program 7.

Program

Program 7: A Python program to override super class constructor and method in sub class.

```
# overriding the base class constructor and method in sub class
class Father:
    def __init__(self):
        self.property = 800000.00

    def display_property(self):
        print('Father\'s property= ', self.property)

class Son(Father):
    def __init__(self):
        self.property = 200000.00

    def display_property(self):
        print('Child\'s property= ', self.property)

# create sub class instance and display father's property
s = Son()
s.display_property()
```

Output:

```
C:\>python inh.py
Child's property= 200000.00
```

In Program 7, in the sub class, we created a constructor and a method with exactly same names as those of super class. When we refer to them, only the sub class constructor and method are executed. The base class constructor and method are not available to the sub class object. That means they are overridden. Overriding should be done when the

programmer wants to modify the existing behavior of a constructor or method in his sub class.

In this case, how to call the super class constructor so that we can access the father's property from the Son class? For this purpose, we should call the constructor of the super class from the constructor of the sub class using the super() method.

The super() Method

super() is a built-in method which is useful to call the super class constructor or methods from the sub class. Any constructor written in the super class is not available to the sub class if the sub class has a constructor. Then how can we initialize the super class instance variables and use them in the sub class? This is done by calling the super class constructor using the super() method from inside the sub class constructor. super() is a built-in method in Python that contains the history of super class methods. Hence, we can use super() to refer to super class constructor and methods from a sub class. So super() can be used as:

```
super().__init__()      # call super class constructor
super().__init__(arguments)# call super class constructor and pass
                           # arguments
super().method()    # call super class method
```

When there is a constructor with parameters in the super class, we have to create another constructor with parameters in the sub class and call the super class constructor using super() from the sub class constructor. In the following example, we are calling the super class constructor and passing 'property' value to it from the sub class constructor.

```
# this is sub class constructor
def __init__(self, property1=0, property=0):
    super().__init__(property)  # send property value to superclass
                                # constructor
    self.property1= property1   # store property1 value into subclass
                                # variable
```

As shown in the preceding code, the sub class constructor has 2 parameters. They are 'property1' and 'property'. So, when we create an object (or instance) to sub class, we should pass two values, as:

```
s = Son(200000.00, 800000.00)
```

Now, the first value 200000 is stored into 'property1' and the second value 800000.00 is stored into 'property'. Afterwards, this 'property' value is sent to super class constructor in the first statement of the sub class constructor. This is shown in Program 8.

Program

Program 8: A Python program to call the super class constructor in the sub class using super().

```
# accessing base class constructor in sub class
class Father:
```

```
    def __init__(self, property=0):
        self.property = property

    def display_property(self):
        print('Father\'s property= ', self.property)

class Son(Father):
    def __init__(self, property1=0, property=0):
        super().__init__(property)
        self.property1= property1

    def display_property(self):
        print('Total property of child= ', self.property1 + self.property)

# create sub class instance and display father's property
s = Son(200000.00, 800000.00)
s.display_property()
```

Output:

```
C:\>python inh.py
Total property of child= 1000000.0
```

To understand the use of super() in a better way, let's write another Python program where we want to calculate areas of a square and a rectangle. Here, we are writing a Square class with one instance variable 'x' since to calculate the area of square, we need one value. Another class Rectangle is derived from Square. So, the value of 'x' is inherited by Rectangle class from Square class. To calculate area of rectangle we need two values. So, we take a constructor with two parameters 'x' and 'y' in the sub class. In this program, we are calling the super class constructor and passing 'x' value as:

```
super().__init__(x)
```

We are also calling super class area() method as:

```
super().area()
```

In this way, super() can be used to refer to the constructors and methods of super class.

Program

Program 9: A Python program to access base class constructor and method in the sub class using super().

```
# Accessing base class constructor and method in the sub class
class Square:
    def __init__(self, x):
        self.x = x

    def area(self):
        print('Area of square= ', self.x*self.x)

class Rectangle(Square):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y
```

```
        def area(self):
            super().area()
            print('Area of rectangle= ', self.x*self.y)

    # find areas of square and rectangle
    a, b = [float(x) for x in input("Enter two measurements: ").split()]
    r = Rectangle(a,b)
    r.area()
```

Output:

```
C:\>python inh.py
Enter two measurements: 10  5.5
Area of square= 100.0
Area of rectangle= 55.0
```

Types of Inheritance

As we have seen so far, the main advantage of inheritance is code reusability. The members of the super class are reusable in the sub classes. Let's remember that all classes in Python are built from a single super class called 'object'. If a programmer creates his own classes, by default object class will become super class for them internally. This is the reason, sometimes while creating any new class, we mention the object class name in parentheses as:

```
class Myclass(object):
```

Here, we are indicating that object is the super class for Myclass. Of course, writing object class name is not mandatory and hence the preceding code is equivalent to writing:

```
class Myclass:
```

Now, coming to the types of inheritance, there are mainly 2 types of inheritance available. They are:

- Single inheritance
- Multiple inheritance

Single Inheritance

Deriving one or more sub classes from a single base class is called 'single inheritance'. In single inheritance, we always have only one base class, but there can be n number of sub classes derived from it. For example, 'Bank' is a single base class from where we derive 'AndhraBank' and 'StateBank' as sub classes. This is called single inheritance. Consider Figure 14.3. It is convention that we should use the arrow head towards the base class (i.e. super class) in the inheritance diagrams.

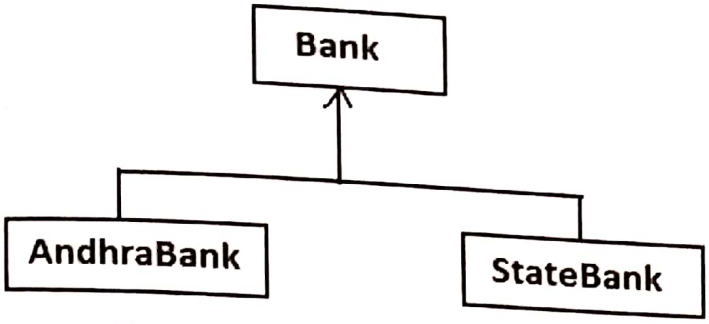

Figure 14.3: Single Inheritance Example

In Program 10, we are deriving two sub classes AndhraBank and StateBank from the single base class, i.e. Bank. All the members (i.e. variables and methods) of Bank class will be available to the sub classes. In the Bank class, we have some 'cash' variable and a method to display that, as:

```
class Bank(object):
    cash = 100000000
    @classmethod
    def available_cash(cls):
        print(cls.cash)
```

Here, the class variable 'cash' is declared in the class and initialized to 10 crores. The available_cash() is a class method that is accessing this variable as 'cls.cash'. When we derive AndhraBank class from Bank class as:

```
class AndhraBank(Bank):
```

The 'cash' variable and available_cash() methods are accessible to AndhraBank class and we can use them inside this sub class. Similarly, we can derive another sub class by the name StateBank from Bank class as:

```
class StateBank(Bank):
    cash = 20000000   # class variable in the present sub class
    @classmethod
    def available_cash(cls):
        print(cls.cash + Bank.cash)
```

Here, StateBank has its own class variable 'cash' that contains 2 crores. So, the total cash available to StateBank is 10 crores + 2 crores = 12 crores. Please observe the last line in the preceding code:

```
print(cls.cash + Bank.cash)
```

Here, 'cls.cash' represents the current class's 'cash' variable and 'Bank.cash' represents the Bank base class 'cash' variable.

Program

Program 10: A Python program showing single inheritance in which two sub classes are derived from a single base class.

```
# single inheritance
class Bank(object):
cash = 100000000
    @classmethod
```

```
    def available_cash(cls):
        print(cls.cash)

class AndhraBank(Bank):
    pass

class StateBank(Bank):
    cash = 20000000
    @classmethod
    def available_cash(cls):
        print(cls.cash + Bank.cash)

a = AndhraBank()
a.available_cash()

s = StateBank()
s.available_cash()
```

Output:

```
C:\>python inh.py
100000000
120000000
```

Multiple Inheritance

Deriving sub classes from multiple (or more than one) base classes is called 'multiple inheritance'. In this type of inheritance, there will be more than one super class and there may be one or more sub classes. All the members of the super classes are by default available to sub classes and the sub classes in turn can have their own members. The syntax for multiple inheritance is shown in the following statement:

```
class Subclass(Baseclass1, Baseclass2, … ):
```

The best example for multiple inheritance is that parents producing the children and the children inheriting the qualities of the parents. Consider Figure 14.4. Suppose, Father and Mother are two base classes and Child is the sub class derived from these two base classes. Now, whatever the members are found in the base classes are available to the sub class. For example, the Father class has a method that displays his height as 6.0 foot and the Mother class has a method that displays her color as brown. To make the Child class acquire both these qualities, we have to make it a sub class for both the Father and Mother class. This is shown in Program 11.

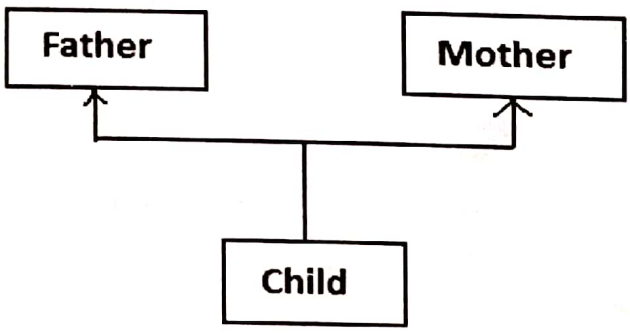

Figure 14.4: Multiple inheritance example

Program

Program 11: A Python program to implement multiple inheritance using two base classes.

```
# multiple inheritance
class Father:
    def height(self):
        print('Height is 6.0 foot')

class Mother:
    def color(self):
        print('Color is brown')

class Child(Father, Mother):
    pass

c = Child()
print('Child\'s inherited qualities: ')
c.height()
c.color()
```

Output:

```
C:\>python inh.py
Child's inherited qualities:
Height is 6.0 foot
Color is brown
```

Problems in Multiple Inheritance

If the sub class has a constructor, it overrides the super class constructor and hence the super class constructor is not available to the sub class. But writing constructor is very common to initialize the instance variables. In multiple inheritance, let's assume that a sub class 'C' is derived from two super classes 'A' and 'B' having their own constructors. Even the sub class 'C' also has its constructor. To derive C from A and B, we write:

```
class C(A, B):
```

Also, in class C's constructor, we call the super class super class constructor using super().__init__(). Now, if we create an object of class C, first the class C constructor is called. Then super().__init__() will call the class A's constructor. Consider Program 12.

Program

Program 12: A Python program to prove that only one class constructor is available to sub class in multiple inheritance.

```
# when super classes have constructors
class A(object):
    def __init__(self):
        self.a = 'a'
        print(self.a)
```

```
class B(object):
    def __init__(self):
        self.b = 'b'
        print(self.b)

class C(A, B):
    def __init__(self):
        self.c = 'c'
        print(self.c)
        super().__init__()

# access the super class instance vars from C
o = C()   # o is object of class C
```

Output:

```
C:\>python inh.py
c
a
```

The output of the program indicates that when class C object is created the C's constructor is called. In class C, we used the statement: super().__init__() that calls the class A's constructor only. Hence, we can access only class A's instance variables and not that of class B. In Program 12, we created sub class C, as:

```
class C(A, B):
```

This means class C is derived from A and B as shown in the Figure 14.5. Since all classes are sub classes of object class internally, we can take classes A and B are sub classes of object class.

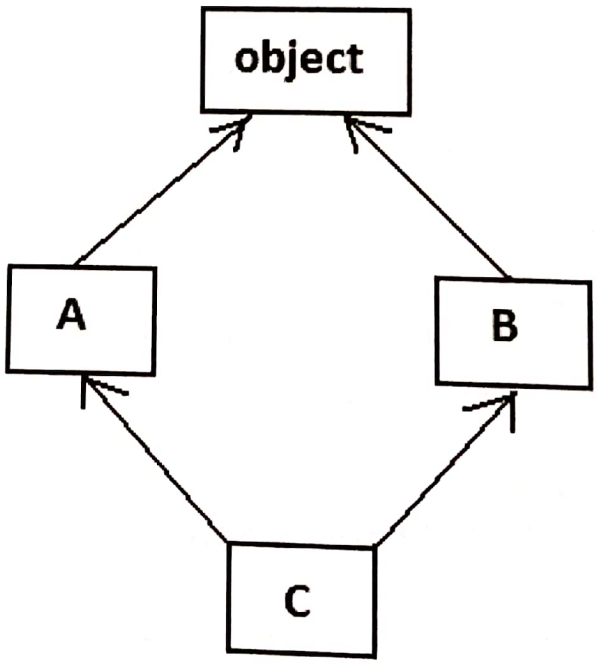

Figure 14.5: The effect of class C(A, B)

In the above figure, class A is at the left side and class B is at the right side for the class C. The searching of any attribute or method will start from the sub class C. Hence, C's constructor is accessed first. As a result, it will display 'c'. Observe the code in C's constructor:

```
    def __init__(self):
        self.c = 'c'
        print(self.c)
        super().__init__()
```

The last line in the preceding code, i.e. super().__init__() will call the constructor of the class which is at the left side. So, class A's constructor is executed and 'a' is displayed. If class A does not have a constructor, then it will call the constructor of the right hand side class, i.e. B. But since class A has a constructor, the search stopped here.

If the class C is derived as:

```
class C(B, A):
```

Then the output will be:

```
c
b
```

The problem we should understand is that the class C is unable to access constructors of both the super classes. It means C cannot access all the instance variables of both of its super classes. If C wants to access instance variables of both of its super classes, then the solution is to use super().__init__() in every class. This is shown in Program 13.

Program

Program 13: A Python program to access all the instance variables of both the base classes in multiple inheritance.

```
# when super classes have constructors - v2.0
class A(object):
    def __init__(self):
        self.a = 'a'
        print(self.a)
        super().__init__()

class B(object):
    def __init__(self):
        self.b = 'b'
        print(self.b)
        super().__init__()

class C(A, B):
    def __init__(self):
        self.c = 'c'
        print(self.c)
        super().__init__()

# access the super class instance vars from C
o = C()   # o is object of class C
```

Output:

```
C:\>python inh.py
c
a
b
```

We will apply the diagram given in Figure 14.5 to Program 13. The search will start from C. As the object of C is created, the constructor of C is called and 'c' is displayed. Then super().__init__() will call the constructor of left side class, i.e. of A. So, the constructor of A is called and 'a' is displayed. But inside the constructor of A, we again called its super class constructor using super().__init__(). Since 'object' is the super class for A, an attempt to execute object class constructor will be done. But object class does not have any constructor. So, the search will continue down to right hand side class of object class. That is class B. Hence B's constructor is executed and 'b' is displayed. After that the statement super().__init__() will attempt to execute constructor of B's super class. That is nothing but 'object' class. Since object class is already visited, the search stops here. As a result the output will be 'c', 'a', 'b'. Searching in this manner for constructors or methods is called *Method Resolution Order(MRO)*.

Method Resolution Order (MRO)

In the multiple inheritance scenario, any specified attribute or method is searched first in the current class. If not found, the search continues into parent classes in depth-first, left to right fashion without searching the same class twice. Searching in this way is called Method Resolution Order (MRO). There are three principles followed by MRO.

- The first principle is to search for the sub class before going for its base classes. Thus if class B is inherited from A, it will search B first and then goes to A.
- The second principle is that when a class is inherited from several classes, it searches in the order from left to right in the base classes. For example, if class C is inherited from A and B as class C(A,B), then first it will search in A and then in B.
- The third principle is that it will not visit any class more than once. That means a class in the inheritance hierarchy is traversed only once exactly.

Understanding MRO gives us clear idea regarding which classes are executed and in which sequence. We can easily estimate the output when several base classes are involved. To know the MRO, we can use mro() method as:

```
Classname.mro()
```

This returns the sequence of execution of the classes, starting from the class with which the method is called. As depicted in Figure 14.6, we are going to create inheritance hierarchy with several classes. The sub class for all these classes is P. This is shown in Program 14.

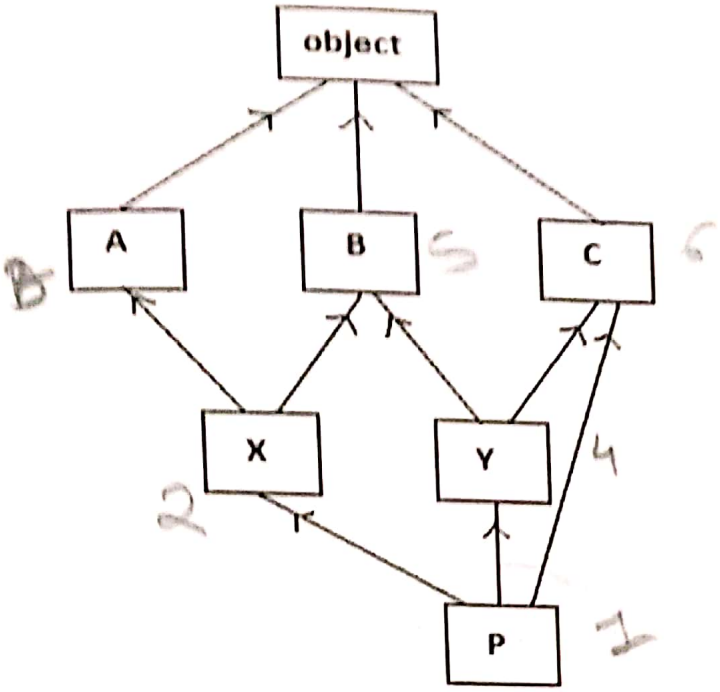

Figure 14.6: Inheritance hierarchy with several classes

Program 14: A Python program to understand the order of execution of methods in several base classes according to MRO.

```
# multiple inheritance with several classes
class A(object):
    def method(self):
        print('A class method')
        super().method()

class B(object):
    def method(self):
        print('B class method')
        super().method()

class C(object):
    def method(self):
        print('C class method')

class X(A, B):
    def method(self):
        print('X class method')
        super().method()

class Y(B, C):
    def method(self):
        print('Y class method')
        super().method()
```

```
class P(X,Y,C):
    def method(self):
        print('P class method')
        super().method()
p = P()
p.method()
```

Output:

```
C:\>python inh.py
P class method
X class method
A class method
Y class method
B class method
C class method
```

To understand the sequence of execution, we should apply MRO. The sub class at the bottom-most level is P. So, first P class method is executed (See output line 1). This class is derived from 3 base classes in the order of X, Y, and C. Hence from left to right, P's first base class X is searched (See output line 2). But, X is derived from 2 more base classes in the order of A, B. Hence, A is searched (See output line 3). Since A does not have a user-defined super class, then it comes down to the class P's second base class, i.e. Y (See output line 4). But Y is derived from two more base classes in the order B, C. Hence from left to right, it searches in B first (See output line 5). Since class B does not have a user-defined super class, the search comes back to the 3rd base class of class P, i.e. class C (output line 6).

If we use mro() method on class P as:

```
print(P.mro())
```

It will display the following output which can be matched with our program output:

```
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>,
<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class
'object'>]
```

Polymorphism

Polymorphism is a word that came from two Greek words, *poly* means many and *morphos* means forms. If something exhibits various forms, it is called polymorphism. Let's take a simple example in our daily life. Assume that we have wheat flour. Using this wheat flour, we can make burgers, rotis, or loaves of bread. It means same wheat flour is taking different edible forms and hence we can say wheat flour is exhibiting polymorphism. Consider Figure 14.7:

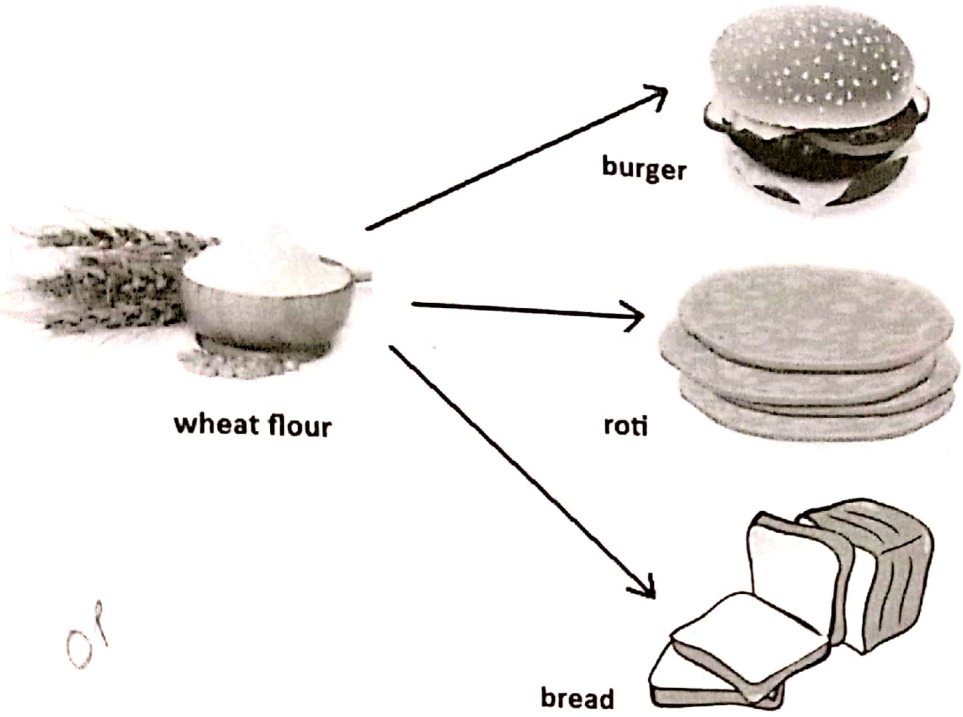

Figure 14.7: Polymorphism where wheat flour takes different edible forms

In programming, a variable, object or a method will also exhibit the same nature as that of the wheat flour. A variable may store different types of data, an object may exhibit different behaviors in different contexts or a method may perform various tasks in Python. This type of behavior is called polymorphism. So, how can we define polymorphism? If a variable, object or method exhibits different behavior in different contexts, it is called polymorphism. Python has built-in polymorphism. The following topics are examples for polymorphism in Python:

- Duck typing philosophy of Python
- Operator overloading
- Method overloading
- Method overriding

Duck Typing Philosophy of Python

We know that in Python, the data type of the variables is not explicitly declared. This does not mean that Python variables do not have a type. Every variable or object in Python has a type and the type is implicitly assigned depending on the purpose for which the variable is used. In the following examples, 'x' is a variable. If we store integer into that variable, its type is taken as 'int' and if we store a string into that variable, its type is taken as 'str'. To check the type of a variable or object, we can use type() function.

```
x = 5   # store integer into x
print(type(x))   # display type of x
<class 'int'>

x = 'Hello'   # store string into x
print(type(x))   # display type of x
<class 'str'>
```

Python variables are names or tags that point to memory locations where data is stored. They are not worried about which data we are going to store. So, if 'x' is a variable, we can make it refer to an integer or a string as shown in the previous examples. We can conclude two points from this discussion:

1. Python's type system is 'strong' because every variable or object has a type that we can check with the type() function.
2. Python's type system is 'dynamic' since the type of a variable is not explicitly declared, but it changes with the content being stored.

Similarly, if we want to call a method on an object, we do not need to check the type of the object and we do not need to check whether that method really belongs to that object or not. For example, take a method call_talk() that accepts an object (or instance) .

```
def call_talk(obj):
    obj.talk()
```

The call_talk() method is receiving an object 'obj' from outside and using this object, it is invoking (or calling) talk() method. It is not required to mention the type of the object 'obj' or to check whether the talk() method belongs to that object or not. If the object passed to this method belongs to Duck class, then talk() method of Duck class is called. If the object belongs to Human class, then the talk() method of Human class is called. This is how Python understands. This is shown in Program 15.

Program

Program 15: A Python program to invoke a method on an object without knowing the type (or class) of the object.

```
# duck typing example
# Duck class contains talk() method
class Duck:
    def talk(self):
        print('Quack, quack!')

# Human class contains talk() method
class Human:
    def talk(self):
        print('Hello, hi!')

# this method accepts an object and calls talk() method
def call_talk(obj):
    obj.talk()

# call call_talk() method and pass an object
# depending on type of object, talk() method is executed
x = Duck()
```

```
call_talk(x)
x = Human()
call_talk(x)
```

Output:

```
C:\>python duck.py
Quack, quack!
Hello, hi!
```

The idea presented in Program 15 is that we do not need a type in order to invoke an existing method on an object. If the method is defined on the object, then it can be called. Thus, when we passed Duck object to call_talk(), it has called the talk() method of Duck type (i.e. Duck class). When we passed Human object to call_talk(), it has called talk() method of Human type.

During runtime, if it is found that the method does not belong to that object, there will be an error called 'AttributeError'. This is shown in Program 16.

Program

Program 16: A Python program to call a method that does not appear in the object passed to the method.

```
# duck typing example - v2.0
# Dog class contains bark() method
class Dog:
    def bark(self):
        print('Bow, wow!')

# Duck class contains talk() method
class Duck:
    def talk(self):
        print('Quack, quack!')

# Human class contains talk() method
class Human:
    def talk(self):
        print('Hello, hi!')

# this method accepts an object and calls talk() method
def call_talk(obj):
    obj.talk()

# call call_talk() method and pass an object
# depending on type of object, talk() method is executed
x = Duck()
call_talk(x)
x = Human()
call_talk(x)
x = Dog()
call_talk(x)   # ERROR occurs in this call
```

Output:

```
C:\>python duck1.py
Quack, quack!
Hello, hi!
Traceback (most recent call alst):
  File "duck1.py", line 27, in <module>
    call_talk(x)
  File "duck1.py", line 18, in call_talk
    obj.talk()
AttributeError: 'Dog' object has no attribute 'talk'
```

In Program 16, we are passing Dog class object to call_talk() method in the last statement. Using this object, call_talk() method called the talk() method as: obj.talk(). Since the object type is Dog, this call to talk() method tries to execute talk() method of Dog class which was not found. Hence there was an error.

In the preceding program, let's observe the call_talk() method that accepts an object 'obj' and calls talk() method on the object.

```
def call_talk(obj):
    obj.talk()   # call talk() method of object
```

This method is calling talk() method of the object 'obj'. It is not bothered about which class object it is. We can pass any class object as long as that object contains the talk() method. That means we can pass Duck object or Human object since they contain the talk() method. But when we pass the Dog object, there would be an error since it does not contain talk() method.

So in Python, we never worry about the type (class) of objects. The object type is distinguished only at runtime. If 'it walks like a duck and talks like a duck, it must be a duck' – this is the principle we follow. This is called *duck typing*. From the previous example, we can understand that the behavior of the talk() method is changing depending on the object type. This is an example for polymorphism of methods.

We can rewrite Program 16 where we can check whether the object passed to the call_talk() method has the method that is being invoked or not. This is done by rewriting the method as:

```
def call_talk(obj):
    if hasattr(obj, 'talk'):   # if obj has talk() method then
        obj.talk()             # call it on the object
    elif hasattr(obj, 'bark'): # if obj has bark() method then
        obj.bark()   # call it
```

In the preceding code, we are checking whether the object has a method or not with the help of hasattr() function. This function is written in the form of:

```
hasattr(object, attribute)
```

Here, 'attribute' may be a method or variable. If it is found in the object (i.e. in the class to which the object belongs) then this method returns True, else False. Checking the object type (or class) in this manner is called 'strong typing'. Please understand that this is not duck typing.

Program

Program 17: A Python program to check the object type to know whether the method exists in the object or not.

```
# strong typing example
# Dog class contains bark() method
class Dog:
    def bark(self):
        print('Bow, wow!')

# Duck class contains talk() method
class Duck:
    def talk(self):
        print('Quack, quack!')

# Human class contains talk() method
class Human:
    def talk(self):
        print('Hello, hi!')

# this method accepts an object and calls talk() method
def call_talk(obj):
    if hasattr(obj, 'talk'):
        obj.talk()
    elif hasattr(obj, 'bark'):
        obj.bark()
    else:
        print('Wrong object passed...')

# call call_talk() method and pass an object
# depending on type of object, talk() method is executed
x = Duck()
call_talk(x)
x = Human()
call_talk(x)
x = Dog()
call_talk(x)
```

Output:

```
C:\>python strong.py
Quack, quack!
Hello, hi!
Bow, wow!
```

Operator Overloading 3

We know that an operator is a symbol that performs some action. For example, '+' is an operator that performs addition operation when used on numbers. When an operator can perform different actions, it is said to exhibit polymorphism.

Program

Program 18: A Python program to use addition operator to act on different types of objects.

```
# overloading the + operator
# using + on integers to add them
print(10+15)

# using + on strings to concatenate them
s1 = "Red"
s2 = "Fort"
print(s1+s2)

# using + on lists to make a single list
a = [10, 20, 30]
b = [5, 15, -10]
print(a+b)
```

Output:

```
C:\>python op.py
25
RedFort
[10, 20, 30, 5, 15, -10]
```

In Program 18, the '+' operator is first adding two integer numbers. Then the same operator is concatenating two strings. Finally, the same operator is combining two lists of elements and making a single list. In this way, if any operator performs additional actions other than what it is meant for, it is called *operator overloading*. Operator overloading is an example for polymorphism.

Normally, addition operator '+' adds two numbers. For example, we can write 10+15. But we cannot use the addition operator to add two objects, as: obj1+obj2. Our intention of writing like this is to use the addition operator to add the data of these two objects. This is not possible. See the example program 19. In this program, we have two classes 'BookX' and 'BookY'. Each book has a number of pages which is given at the time of creating the objects as:

```
b1 = BookX(100)
b2 = BookY(150)
```

Now, if we write b1+b2, the addition operator cannot add the number of pages which are available in the objects since this operator cannot act on the objects.

Program

Program 19: A Python program to use addition operator to add the contents of two objects.

```
# Using + operator on objects
class BookX:
    def __init__(self, pages):
        self.pages = pages

class BookY:
```

```
    def __init__(self, pages):
        self.pages = pages

b1 = BookX(100)
b2 = BookY(150)
print('Total pages= ', b1+b2)
```

Output:

```
C:\>python op.py
Traceback (most recent call last):
  File "op.py", line 12, in <module>
    print('Total pages= ', b1+b2)
TypeError: unsupported operand type(s) for +: 'BookX' and 'BookY'
```

We can overload the '+' operator to act upon the two objects and perform addition operation on the contents of the objects. That means we are giving additional task to the '+' operator. This comes under operator overloading.

Let's go a bit into internal details. The '+' operator is in fact internally written as a special method, i.e. __add__(). So, if we add two numbers by writing a+b, the internal method is called as: a.__add__(b). By overriding this method to act upon objects, we can make the '+' operator to successfully act on the objects also. Since we want to override, we have to write the method with same name but with objects as:

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

Here, self.a represents the numeric content of first class and other.b represents the numeric content of second class. The preceding method should be written in the first class. To add the pages of BookX and BookY objects, we have to write this method as:

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

Program

Program 20: A Python program to overload the addition operator (+) to make it act on class objects.

```
# overloading + operator to act on objects
class BookX:
    def __init__(self, pages):
        self.pages = pages

    def __add__(self, other):
        return self.pages+other.pages
class BookY:
    def __init__(self, pages):
        self.pages = pages

b1 = BookX(100)
b2 = BookY(150)
print('Total pages= ', b1+b2)
```

Output:

```
C:\>python op1.py
Total pages= 250
```

Table 14.1 summarizes important operators and their corresponding internal methods that can be overridden to act on objects. These methods are called *magic methods*.

Table 14.1: Operators and Corresponding Magic Methods

Operator	Magic method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
/	object.__div__(self, other)
//	object.__floordiv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other[, modulo])
**=	object.__ipow__(self, other)
<	object.__lt__(self, other)
<=	object.__le__(self, other)
>	object.__gt__(self, other)
>=	object.__ge__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)

We will plan another program where we want to overload the greater than (>) operator. This operator is normally used on numbers to compare them. It returns True of False depending on the result. But if want to use it on objects, we have to overload it. For this

purpose the magic method __gt__() should be overridden. For example, to compare the pages of two books, we can write this method as:

```
def __gt__(self, other):
    return self.pages>other.pages
```

The preceding method returns True if the pages in first object is greater than those of second object, otherwise False.

Program

Program 21: A Python program to overload greater than (>) operator to make it act on class objects.

```
# overloading > operator
class Ramayan:
    def __init__(self, pages):
        self.pages = pages

    def __gt__(self, other):
        return self.pages>other.pages

class Mahabharat:
    def __init__(self, pages):
        self.pages = pages

b1 = Ramayan(1000)
b2 = Mahabharat(1500)
if(b1>b2):
    print('Ramayan has more pages')
else:
    print('Mahabharat has more pages')
```

Output:

```
C:\>python op2.py
Mahabharat has more pages
```

Another example is where we have an Employee class that contains the name and daily salary of the employee. Another class Attendance contains the employee name and his number of working days. To get the total salary of the employee we have to multiply the daily salary with the number of days worked. That means we have to multiply the Employee object data with Attendance object data. For this purpose, we should overload the multiplication operator.

Since multiplication operator is internally represented by the magic method __mul__(), we have to rewrite or override this method to make it act on the objects as:

```
def __mul__(self, other):
    return self.salary*other.days
```

When * operator is used on the objects, this method is called and the required result is obtained.

Program

Program 22: A Python program to overload the multiplication (*) operator to make it act on objects.

```
# overloading the * operator
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

x1 = Employee('Srinu', 500.00)
x2 = Attendance('Srinu', 25)
print('This month salary= ', x1*x2)
```

Output:

```
C:\>python op3.py
This month salary= 12500.0
```

Method Overloading

If a method is written such that it can perform more than one task, it is called *method overloading.* We see method overloading in the languages like Java. For example, we call a method as:

```
sum(10, 15)
sum(10, 15, 20)
```

In the first call, we are passing two arguments and in the second call, we are passing three arguments. It means, the sum() method is performing two distinct operations: finding sum of two numbers or sum of three numbers. This is called method overloading. In Java, to achieve this, we write two sum() methods with different number of parameters as:

```
sum(int a, int b) {}
sum(int a, int b, int c) {}
```

Since same name is given for these two methods, the user feels that the same method is performing the two operations. This is how the method overloading is done in Java.

Method overloading is not available in Python. Writing more than one method with the same name is not possible in Python. So, 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. For example, we can write a sum() method with default value 'None' for the arguments as:

```
def sum(self, a=None, b=None, c=None):
    if a!=None and b!=None and c!=None:
        print('Sum of three= ', a+b+c)
    elif a!=None and b!=None:
        print('Sum of two= ', a+b)
```

Here, sum() has three arguments 'a','b','c' whose values by default are 'None'. This 'None' indicates nothing or no value and similar to 'null' in languages like Java. While calling this method, if the user enters three values, then the arguments: 'a',' b' and 'c' will not be 'None'. They get the values entered by the user. If the user enters only two values, then the first two arguments 'a' and 'b' only will take those values and the third argument 'c' will become 'None'. In this way, it is possible to find sum of either two numbers or three numbers.

Program

Program 23: A Python program to show method overloading to find sum of two or three numbers.

```
# method overloading
class Myclass:
    def sum(self, a=None, b=None, c=None):
        if a!=None and b!=None and c!=None:
            print('Sum of three= ', a+b+c)
        elif a!=None and b!=None:
            print('Sum of two= ', a+b)
        else:
            print('Please enter two or three arguments')

# call sum() using object
m = Myclass()
m.sum(10, 15, 20)
m.sum(10.5, 25.55)
m.sum(100)
```

Output:

```
C:\>python over.py
Sum of three= 45
Sum of two= 36.05
Please enter two or three arguments
```

In Program 23, the sum() method is calculating sum of two or three numbers and hence it is performing more than one task. Hence it is an overloaded method. In this way, overloaded methods achieve polymorphism.

Method Overriding 5.

We already discussed constructor overriding and method overriding under inheritance section. When there is a method in the super class, writing the same method in the sub class so that it replaces the super class method is called 'method overriding'. The programmer overrides the super class methods when he does not want to use them in sub class. Instead, he wants a new functionality to the same method in the sub class.

In inheritance, if we create super class object (or instance), we can access all the members of the super class but not the members of the sub class. But if we create sub class object, then both the super class and sub class members are available since the sub class object contains a copy of the super class. Hence, in inheritance we always create sub class object.

In Program 24, we created Square class with the area() method that calculates the area of the square. Circle class is a sub class to Square class that contains the area() method rewritten with code to calculate area of circle. When we create sub class object as:

```
c = Circle()   # create sub class object
c.area(15)   # call area() method
```

Then the area() method of Circle class is called but not the area() method of Square class. The reason is that the area() method of sub class has overridden the area() method of the super class.

Program

Program 24: A Python program to override the super class method in sub class.

```
# method overriding
import math
class Square:
    def area(self, x):
        print('Square area= %.4f}'% x*x)

class Circle(Square):
    def area(self, x):
        print('Circle area= %.4f'% (math.pi*x*x))

# call area() using sub class object
c = Circle()
c.area(15)
```

Output:

```
C:\>python over.py
Circle area= 706.8583
```

Calling the area() method using Circle class object will execute Circle class area() method. But in case, the programmer wants to calculate the area of the square, he can call the same area() method using the Square class object. So, same area() method is performing two different tasks depending on the object type. This is an example for polymorphism.

Polymorphism is a powerful feature in all object oriented programming languages. We will observe polymorphism in case of abstract classes and interfaces in the next chapter.

Points to Remember

- The concept of deriving new classes from the existing classes such that the new classes inherit all the members of the existing classes is called inheritance.

- The already existing class is called super class or base class and the newly created class is called sub class or derived class.
- The syntax for inheritance is:

  class Subclass(Baseclass):
- The main advantage of inheritance is code reusability. This will increase the productivity of the organization.
- Since the sub class contains a copy of super class, all the members of the super class are available to the sub class.
- The variables, methods and also constructors of the super class are available to the sub class.
- Writing a constructor in the sub class will override the constructor of the super class. In that case, the super class constructor will not be available.
- Writing a method in the sub class with the same name as that of the super class method is called method overriding. In this case, the super class method is not available. Only the sub class method is executed always.
- super() is a method that is useful to refer to all members of the super class from a sub class. For example,

  super().__init__() # call super class constructor

  super().__init__(arguments) # call super class constructor and pass arguments

  super().method() # call super class method
- Python supports single inheritance and multiple inheritance and all other forms of inheritance which may rise from combining these two types of inheritances.
- Deriving one or more sub classes from a single base class is called 'single inheritance'.
- Deriving sub classes from multiple (or more than one) base classes is called 'multiple inheritance'.
- Starting from the current class, searching in parent classes in depth-first, left to right fashion without searching the same class twice is called Method Resolution Order (MRO).
- If a variable, object or method exhibits different behavior in different contexts, it is called polymorphism.
- Python follows duck typing where the type of the object is not checked while invoking the method on the object. Any object is accepted as long as that method is found in the object.

- If any operator performs additional actions other than what it is meant for, it is called operator overloading. Operator overloading is an example for polymorphism.
- Magic methods are the methods which are internal representations of operator symbols.
- If a method is written such that it can perform more than one task, it is called method overloading. Method overloading is an example for polymorphism.
- When there is a method in the super class, writing the same method in the sub class so that it replaces the super class method is called method overriding. Method overriding is an example of polymorphism.

ABSTRACT CLASSES AND INTERFACES

CHAPTER 15

We know that a class is a model for creating objects (or instances). A class contains attributes and actions. Attributes are nothing but variables and actions are represented by methods. When objects are created to a class, the objects also get the variables and actions mentioned in the class. The rule is that anything that is written in the class is applicable to all of its objects. If a method is written in the class, it is available to all the class objects. For example, take a class Myclass that contains a method calculate() that calculates square value of a given number. If we create three objects to this class, all the three objects get the copy of this method and hence, from any object, we can call and use this method. Consider Program 1.

Program

Program 1: A Python program to understand that Myclass method is shared by all of its objects.

```
# A class with a method
class Myclass:
    def calculate(self, x):
        print('Square value= ', x*x)

# all objects share same calculate() method
obj1 = Myclass()
obj1.calculate(2)

obj2 = Myclass()
obj2.calculate(3)

obj3 = Myclass()
obj3.calculate(4)
```

Output:

```
C:\>python ex.py
Square value= 4
Square value= 9
Square value= 16
```

Of course, in the preceding program, the requirement of all the objects is same, i.e., to calculate square value. Then this program is alright. But, sometimes the requirement of the objects will be different and entirely dependent on the specific object only. For example, in the preceding program, if the first object wants to calculate square value, the second object wants the square root value and the third object wants cube value. In such a case, how to write the calculate() method in Myclass?

Since, the calculate() method has to perform three different tasks depending on the object, we cannot write the code to calculate square value in the body of calculate() method. On the other hand, if we write three different methods like calculate_square(), calculate_sqrt(), and calculate_cube() in Myclass, then all the three methods are available to all the three objects which is not advisable. When each object wants one method, providing all the three does not look reasonable. To serve each object with the one and only required method, we can follow the steps:

1. First, let's write a calculate() method in Myclass. This means every object wants to calculate something.
2. If we write body for calculate() method, it is commonly available to all the objects. So let's not write body for calculate() method. Such a method is called abstract method. Since, we write abstract method in Myclass, it is called abstract class.
3. Now derive a sub class Sub1 from Myclass, so that the calculate() method is available to the sub class. Provide body for calculate() method in Sub1 such that it calculates square value. Similarly, we create another sub class Sub2 where we write the calculate() method with body to calculate square root value. We create the third sub class Sub3 where we write the calculate() method to calculate cube value. This hierarchy is shown in Figure 15.1.
4. It is possible to create objects for the sub classes. Using these objects, the respective methods can be called and used. Thus, every object will have its requirement fulfilled.

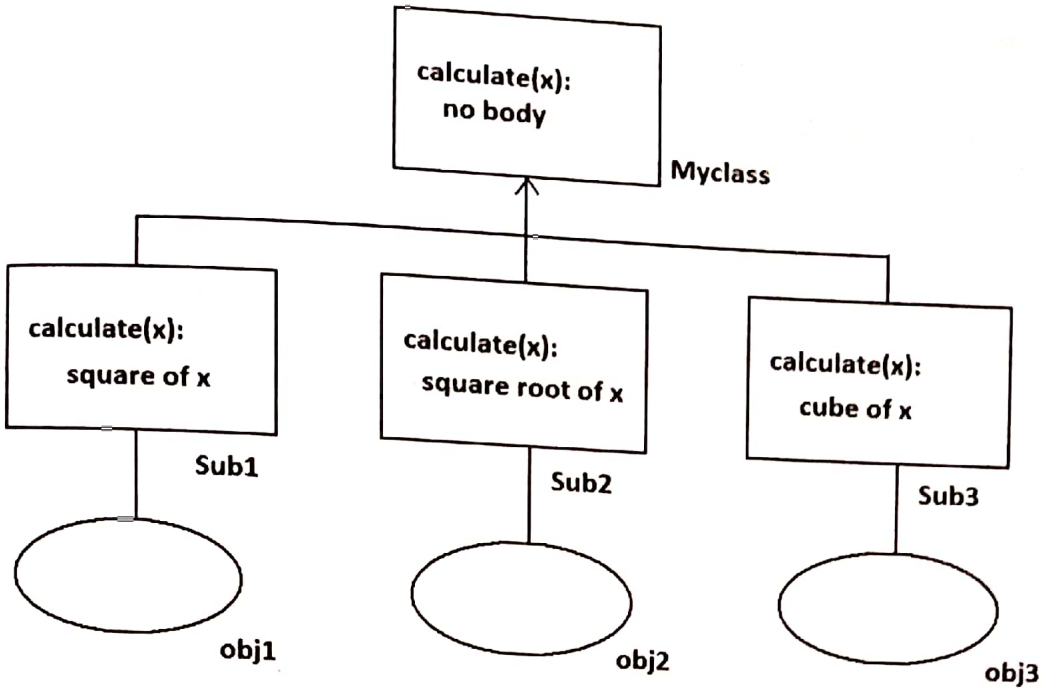

Figure 15.1: Defining a Method in Sub Classes to Suit the Objects

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. Generally abstract methods are written without body since their body will be defined in the sub classes anyhow. But it is possible to write an abstract method with body also. To mark a method as abstract, we should use the decorator @abstractmethod. On the other hand, a concrete method is a method with body.

An abstract class is a class that generally contains some abstract methods. Since, abstract class contains abstract methods whose implementation (or body) is later defined in the sub classes, it is not possible to estimate the total memory required to create the object for the abstract class. So, PVM cannot create objects to 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 objects to the sub classes.

In Program 2, we create Myclass as an abstract super class with an abstract method calculate(). This method does not have any body within it. The way to create an abstract class is to derive it from a meta class ABC that belongs to *abc* (abstract base class) module as:

```
class Abstractclass(ABC):
```

Since all abstract classes should be derived from the meta class ABC which belongs to *abc* (abstract base class) module, we should import this module into our program. A meta class is a class that defines the behavior of other classes. The meta class ABC defines

that the class which is derived from it becomes an abstract class. To import abc module's ABC class and abstractmethod decorator we can write as follows:

```
from abc import ABC, abstractmethod
```

or

```
from abc import *
```

Now, our abstract class 'Myclass' should be derived from the ABC class as:

```
classs Myclass(ABC):
    @abstractmethod
    def calculate(self, x):
        pass   # empty body, no code
```

Observe the preceding code. Our Myclass is abstract class since it is derived from ABC meta class. This class has an abstract method calculate() that does not contain any code. We used @abstractmethod decorator to specify that this is an abstract method. We have to write sub classes where this abstract method is written with its body (or implementation). In Program 2, we are going to write three sub classes: Sub1, Sub2 and Sub3 where this abstract method is implemented as per the requirement of the objects. Since, the same abstract method is implemented differently for different objects, they can perform different tasks.

Program

Program 2: A Python program to create abstract class and sub classes which implement the abstract method of the abstract class.

```
# abstract class example
from abc import ABC, abstractmethod
class Myclass(ABC):
    @abstractmethod
    def calculate(self, x):
        pass   # empty body, no code

# this is sub class of Myclass
class Sub1(Myclass):
    def calculate(self, x):
        print('Square value= ', x*x)

# this is another sub class for Myclass
import math
class Sub2(Myclass):
    def calculate(self, x):
        print('Square root= ', math.sqrt(x))

# third sub class for Myclass
class Sub3(Myclass):
    def calculate(self, x):
        print('Cube value= ', x**3)

# create Sub1 class object and call calculate() method
obj1 = Sub1()
obj1.calculate(16)

# create Sub2 class object and call calculate() method
```

```
obj2 = Sub2()
obj2.calculate(16)
# create Sub3 class object and call calculate() method
obj3 = Sub3()
obj3.calculate(16)
```

Output:

```
C:\>python abs.py
Square value= 256
Square root= 4.0
Cube value= 4096
```

In the preceding program, the same calculate() method written in the abstract class is implemented differently in the three sub classes and it is able to perform different tasks. Since the same method is performing various tasks, it is coming under polymorphism.

Let's take another example to understand the abstract class concept in a better way. We see many cars on the road. These cars are all objects of Car class. For example, Maruti, Santro, Benz are all objects of Car class. Suppose, we plan to write Car class, it contains all the attributes (variables) and actions (methods) of any car object in the world. For example, we can write the following members in Car class:

- **Registration number:** Every car will have a registration number and hence we write this as an instance variable in Car class. All cars whether it is Maruti or Santro should have a registration number. It means registration number is a common feature to all the objects. So, it can be written as an instance variable in the Car class.

- **Fuel tank:** Every car will have a fuel tank, opening and filling the tank is an action. To represent this action, we can write a method like: openTank()

  How do we open and fill the tank? Take the key, open the tank and fill fuel. Let's assume that all cars have same mechanism of opening the tank and filling fuel. So, the code representing the opening mechanism can be written in openTank() method's body. So it becomes a concrete method. A concrete method is a method with body.

- **Steering:** Every car will have a steering wheel and steering the car is an action. For this, we write a method as: steering()

  How do we steer the car? All the cars do not have same mechanism for steering. Maruti cars have manual steering. Santro cars have power steering. So, it is not possible to write a particular mechanism in steering() method. So, this method should be written without body in Car class. Thus, it becomes an abstract method.

- **Brakes:** Every car will have brakes. Applying brakes is an action and hence it can be represented as a method, as: braking()

  How do we apply brakes? All cars do not have same mechanism for brakes. Maruti cars have hydraulic brakes. Santro cars have gas brakes. So, we cannot write a particular braking mechanism in this method. This method, hence, will not have a body in Car class, and hence becomes abstract. See Figure 15.2:

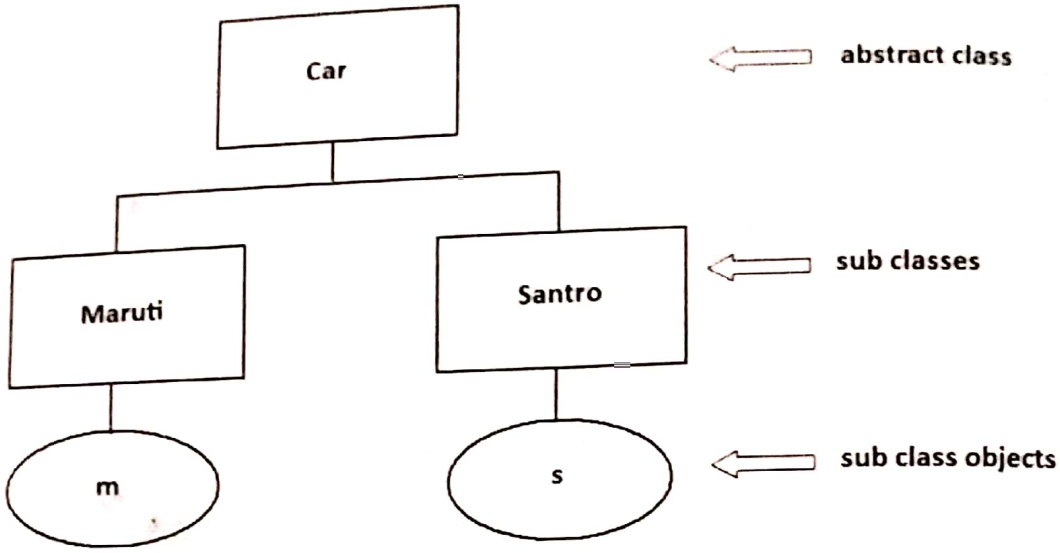

Figure 15.2: Abstract Car Class and its Sub Classes

So, the Car class has an instance variable, one concrete method and two abstract methods. Hence, Car class will become abstract class. See this class in Program 3 given here.

Program

Program 3: A Python program to create a Car abstract class that contains an instance variable, a concrete method and two abstract methods.

```
# This is an abstract class. Save this code as abs.py
from abc import *
class Car(ABC):
    def __init__(self, regno):
        self.regno = regno

    def openTank(self):
        print('Fill the fuel into the tank')
        print('for the car with regno ', self.regno)

    @abstractmethod
    def steering(self):
        pass

    @abstractmethod
    def braking(self):
        pass
```

Output:

```
C:\>python abs.py
C:\>
```

Now that we have written the abstract class, the next step is to derive sub classes from the Car class. In the sub classes, we should take the abstract methods of the Car class and implement (writing body in) them. The reason why we implement the abstract methods in the sub classes is that the implementation of these methods is dependent on

the sub classes. In our program, let's write Maruti and Santro as the two sub classes where the two abstract methods will be implemented accordingly. See these sub classes in Programs 4 and 5.

Program 4: A Python program in which Maruti sub class implements the abstract methods of the super class, Car.

```
# this is a sub class for abstract Car class
from abs import Car
class Maruti(Car):
    def steering(self):
        print('Maruti uses manual steering')
        print('Drive the car')
    def braking(self):
        print('Maruti uses hydraulic brakes')
        print('Apply brakes and stop it')
# create object to Maruti and use its features
m = Maruti(1001)
m.openTank()
m.steering()
m.braking()
```

Output:

```
C:\>python maruti.py
Fill the fuel into the tank
for the car with regno  1001
Maruti uses manual steering
Drive the car
Maruti uses hydraulic brakes
Apply brakes and stop it
```

Program 5: A Python program in which Santro sub class implements the abstract methods of the super class, Car.

```
# this is a sub class for abstract Car class
from abs import Car
class Santro(Car):

    def steering(self):
        print('Santro uses power steering')
        print('Drive the car')

    def braking(self):
        print('Santro uses gas brakes')
        print('Apply brakes and stop it')

# create object to Santro and use its features
s = Santro(7878)
s.openTank()
s.steering()
s.braking()
```

Output:

```
C:\>python santro.py
Fill the fuel into the tank
for the car with regno  7878
Santro uses power steering
Drive the car
Santro uses gas brakes
Apply brakes and stop it
```

An ordinary class can be called rigid class since it can look after the common needs of the objects. There is no scope for catering the individual requirements of objects. Abstract classes are more flexible and useful than ordinary classes since they cater the common needs of the objects as well as their individual needs also.

Interfaces in Python

We learned that an abstract class is a class which contains some abstract methods as well as concrete methods also. Imagine there is a class that contains only abstract methods and there are no concrete methods. It becomes an interface. This means an interface is an abstract class but it contains only abstract methods. None of the methods in the interface will have body. Only method headers will be written in the interface. So an interface can be defined as a specification of method headers. Since, we write only abstract methods in the interface, there is possibility for providing different implementations (body) for those abstract methods depending on the requirements of objects. In the languages like Java, an interface is created using the key word 'interface' but in Python an interface is created as an abstract class only. The interface concept is not explicitly available in Python. We have to use abstract classes as interfaces in Python.

Since an interface contains methods without body, it is not possible to create objects to an interface. In this case, we can create sub classes where we can implement all the methods of the interface. Since the sub classes will have all the methods with body, it is possible to create objects to the sub classes. The flexibility lies in the fact that every sub class can provide its own implementation for the abstract methods of the interface.

Since, none of the methods have body in the interface, we may tend to think that writing an interface is mere waste. This is not correct. In fact, an interface is more useful when compared to the class owing to its flexibility of providing necessary implementation needed by the objects. Let's elucidate this point further with an example. We have some rupees in our hands. We can spend in rupees only by going to a shop where billing is done in rupees. Suppose we have gone to a shop where only dollars are accepted, we cannot use our rupees there. This money is like a 'class'. A class satisfies the only requirement intended for it. It is not useful to handle a different situation.

Suppose we have an international credit card. Now, we can pay by using our credit card in rupees in a shop. If we go to another shop where they expect us to pay in dollars, we can pay in dollars. The same credit card can be used to pay in pounds also. Here, the credit card is like an interface which performs several tasks. In fact, the credit card is a

plastic card and does not hold any money physically. It contains just our name, our bank name and perhaps some number. But how the shop keepers are able to draw the money from the credit card? Behind the credit card, we got our bank account which holds the money from where it is transferred to the shop keepers. This bank account can be taken as a sub class which actually performs the task. See Figure 15.3:

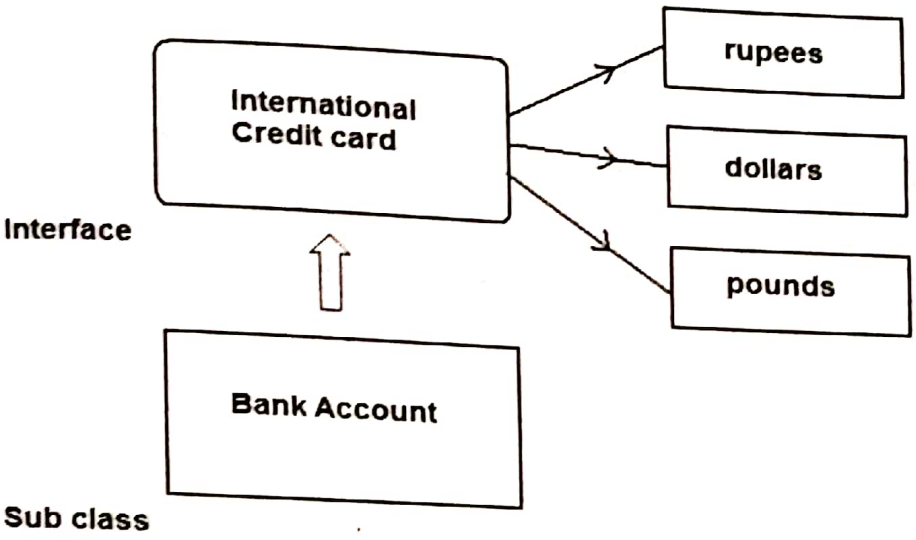

Figure 15.3: Interface and Sub Classes

Let's see how the interface concept is advantageous in software development. A programmer is asked to write a Python program to connect to a database and retrieve the data, process the data and display the results in the form of some reports. For this purpose, the programmer has written a class to connect to Oracle database, something like this:

```
# this class works with Oracle database only
class Oracle:
    def connect(self):
        print('Connecting to Oracle database...')

    def disconnect(self):
        print('Disconnected from Oracle.')
```

This class has a limitation. It can connect only to Oracle database. If a client (user) using any other database (for example, Sybase database) uses this code to connect to his database, this code will not work. So, the programmer is asked to design his code in such a way that it is used to connect to any database in the world. How is it possible?

One way is to write several classes, each to connect to a particular database. Thus, considering all the databases available in the world, the programmer has to write a lot of classes. This takes a lot of time and effort. Even though the programmer spends a lot of time and writes all the classes, by the time the software is released into the market, all the versions of the databases will change and the programmer is supposed to rewrite the classes again to suit the latest versions of databases. This is very cumbersome.

Interface helps to solve this problem. The programmer writes an interface 'Myclass' with abstract methods as shown here:

```
# an interface to connect to any database
class Myclass(ABC):
    @abstractmethod
    def connect(self):
        pass
    @abstractmethod
    def disconnect(self):
        pass
```

We cannot create objects to the interface. So, we need sub classes where all these methods of the interface are implemented to connect to various databases. This task is left for other companies that are called third party vendors. The third party vendors will provide sub classes to Myclass interface. For example, Oracle Corp people may provide a sub class where the code related to connecting to the Oracle database and disconnecting from the database will be provided as:

```
# this is a sub class to connect to Oracle
class Oracle(Myclass):
    def connect(self):
        print('Connecting to Oracle database...')

    def disconnect(self):
        print('Disconnected from Oracle.')
```

Note that 'Oracle' is a sub class of Myclass interface. Similarly, the Sybase company people may provide another implementation class 'Sybase', where code related to connecting to Sybase database and disconnecting from Sybase will be provided as:

```
# this is another sub class to connect to Sybase
class Sybase(Myclass):
    def connect(self):
        print('Connecting to Sybase database...')

    def disconnect(self):
        print('Disconnected from Sybase.')
```

Now, it is possible to create objects to the sub classes and call the connect() method and disconnect() methods from a main program. This main program is also written by the same programmer who develops the interface. Let's understand that the programmer who develops the interface does not know the names of the sub classes as they will be developed in future by the other companies. In the main program, the programmer is supposed to create objects to the sub classes without knowing their names. This is done using globals() function.

First, the programmer should accept the database name from the user. It may be 'Oracle' or 'Sybase'. This name should be taken in a string, say 'str'. The next step is to convert this string into a class name using the built-in function globals(). The globals() function returns a dictionary containing current global names and globals()[str] returns the name of the class that is in 'str'. Hence, we can get the class name as:

```
classname = globals()[str]
```

Now, create an object to this class and call the methods as:

```
x = classname()    # x is object of the class
x.connect()
x.disconnect()
```

The connect() method establishes connection with the particular database and disconnect() method disconnects from the database. The complete program is shown in Program 6.

Program

Program 6: A Python program to develop an interface that connects to any database.

```
# abstract class works like an interface
from abc import *
class Myclass(ABC):
    @abstractmethod
    def connect(self):
        pass
    @abstractmethod
    def disconnect(self):
        pass

# this is a sub class
class Oracle(Myclass):
    def connect(self):
        print('Connecting to Oracle database...')

    def disconnect(self):
        print('Disconnected from Oracle.')

# this is another sub class
class Sybase(Myclass):
    def connect(self):
        print('Connecting to Sybase database...')

    def disconnect(self):
        print('Disconnected from Sybase.')

class Database:
    # accept database name as a string
    str = input('Enter database name: ')

    # convert the string into classname
    classname = globals()[str]

    # create an object to that class
    x = classname()

    # call the connect() and disconnect() methods
    x.connect()
    x.disconnect()
```

Output:

```
C:\>python inter.py
Enter database name: Oracle
Connecting to Oracle database...
Disconnected from Oracle.
```

```
C:\>python inter.py
Enter database name: Sybase
Connecting to Sybase database...
Disconnected from Sybase.
```

When an interface is created, it is not necessary for the programmer to provide the sub classes for the interface. Any third party vendors can do that. The client or user purchases and uses the interface and the sub class depending on his requirements. If he wants to connect to Oracle, he will purchase Oracle sub class. If he wants to connect to Sybase, he will purchase Sybase sub class.

The question is how the third party vendors know which methods they should write in the sub classes? In this case, API documentation will help them. An API (Application Programming Interface) documentation file is a text file or html file that contains description of all the features of software, language or a product. After developing the software, the programmer creates API documentation file that contains description of all the classes, methods and attributes. This file is referred by the third party vendors to know about the interface and its methods. Then they write the same methods in the sub classes. This entire discussion is represented in Figure 15.4:

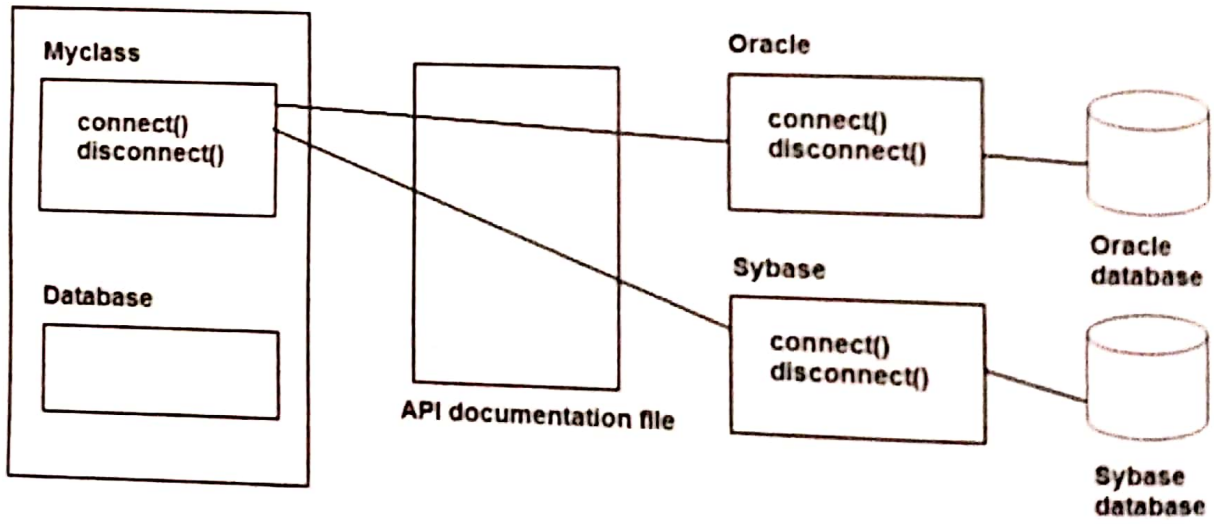

Figure 15.4: Interface Communicating with Different Databases

Let's take another example where interface is used. We want to write Printer interface which is used to send data to different printers. This interface has a method printit() that sends text to the printer and another method disconnect() that disconnects the printer after printing is done. Of course, this program is a model how the printer interface can be used. It does not send the text to a real printer. On the other hand, it displays the text on the screen.

Printer interface is implemented by IBM people such that it sends text to IBM printer. Similarly, Epson people provide a different implementation to the Printer interface such that it sends text to Epson printer. These sub classes are written as IBM and Epson classes.

To use a printer, first of all we should know which printer is used by the client. We assume that the printer name is generally stored in the config.txt file at the time of installing the printer driver software. So, open notepad (or any text editor) and create a file with the name config.txt and store a single line that represents the printer driver name as:

```
Epson
```

And then save the file. Alternately, we can store the name IBM in the config.txt file. The name of the printer available in the config.txt file can be read from the file using readline() method as:

```
with open("config.txt", "r") as f:
    str = f.readline()    # read printer name from f and store into str
```

This reads one line from the file containing the string 'Epson' which was already stored by us in the config.txt file. This 'Epson' would appear in 'str' as a string. Retrieving the class name from this 'str' is done using globals() method. Then we create an object to that class and use it.

Program

Program 7: A Python program which contains a Printer interface and its sub classes to send text to any printer.

```
# An interface to send text to any printer
from abc import *
# create an interface
class Printer(ABC):
    @abstractmethod
    def printit(self, text):
        pass
    @abstractmethod
    def disconnect(self):
        pass

# this is sub class for IBM printer
class IBM(Printer):
    def printit(self, text):
        print(text)

    def disconnect(self):
        print('Printing completed on IBM printer.')

# this is sub class for Epson printer
class Epson(Printer):
    def printit(self, text):
        print(text)

    def disconnect(self):
        print('Printing completed on Epson printer.')

class UsePrinter:
    # accept printer name as a string from configuration file
    with open("config.txt", "r") as f:
        str = f.readline()
```

```
# convert the string into classname
classname = globals()[str]

# create an object to that class
x = classname()

# call the printit() and disconnect() methods
x.printit('Hello, this is sent to printer')
x.disconnect()
```

Output:

```
C:\>python inter1.py
Hello, this is sent to printer
Printing completed on Epson printer.
```

Please remember that we should run this program after creating the config.txt file. If we created the config.txt file with the string 'IBM', then the output of the preceding program would be:

```
Hello, this is sent to printer
Printing completed on IBM printer.
```

We can observe that the same printit() and disconnect() methods when called are performing different tasks in different contexts. They are sending text to Epson printer or IBM printer depending upon the user requirement. This is an example for polymorphism. Let's understand that the abstract classes and interfaces are examples for polymorphic behavior.

Abstract Classes vs. Interfaces

Python does not provide interface concept explicitly. It provides abstract classes which can be used as either abstract classes or interfaces. It is the discretion of the programmer to decide when to use an abstract class and when to go for an interface. Generally, abstract class is written when there are some common features shared by all the objects as they are. For example, take a class WholeSaler which represents a whole sale shop with text books and stationery like pens, papers and note books as:

```
# an abstract class
class WholeSaler(ABC):
    @abstractmethod
    def text_books(self):
        pass
    @abstractmethod
    def stationery(self):
        pass
```

Let's take Retailer1, a class which represents a retail shop. Retailer1 wants text books of X class and some pens. Similarly, Retailer2 also wants text books of X class and some papers. In this case, we can understand that the text_books() is the common feature shared by both the retailers. But the stationery asked by the retailers is different. This means, the stationery has different implementations for different retailers but there is a common feature, i.e., the text books. So in this case, the programmer designs the WholeSaler class as an abstract class. Retailer1 and Retailer2 are sub classes.

On the other hand, the programmer uses an interface if all the features need to be implemented differently for different objects. Suppose, Retailer1 asks for VII class text books and Retailer2 asks for X class text books, then even the text_books() method of WholeSaler class needs different implementations depending on the retailer. It means, the text_books() method and also stationery() methods should be implemented differently depending on the retailer. So, in this case, the programmer designs the WholeSaler as an interface and Retailer1 and Retailer2 become sub classes.

There is a responsibility for the programmer to provide the sub classes whenever he writes an abstract class. This means the same development team should provide the sub classes for the abstract class. But if an interface is written, any third party vendor will take the responsibility of providing sub classes. This means, the programmer prefers to write an interface when he wants to leave the implementation part to the third party vendors.

In case of an interface, every time a method is called, PVM should search for the method in the implementation classes which are installed elsewhere in the system and then execute the method. This takes more time. But when an abstract class is written, since the common methods are defined within the abstract class and the sub classes are generally in the same place along with the software, PVM will not have that much overhead to execute a method. Hence, interfaces are slow when compared to abstract classes.

Points to Remember

- An abstract method is a method whose action is redefined in the sub classes as per the requirement of the objects.
- Generally, an abstract method is written without any body. But it is possible to write an abstract method with body also.
- A method with body is called concrete method.
- An abstract class is a class that contains some abstract methods. An abstract class can also contain concrete methods.
- Abstract methods are written with a decorator @abstractmethod above them.
- It is not possible to create an object (or instance) to an abstract class.
- Every abstract class should derive from ABC class that is available in abc (abstract base class) module.
- All the abstract methods of the abstract class should be rewritten with body in the sub classes. That means the abstract methods should be overridden in the sub classes.
- We can create objects (or instances) to sub classes.

- If a sub class does not implement all the methods of the abstract super class, then that sub class should also be declared as abstract class and an object to that sub class cannot be created.
- An abstract class can perform various tasks through various implementations of its methods in the sub classes.
- An abstract class will become an interface when it contains only abstract methods and there are no concrete methods.
- It is not possible to create objects (or instances) to interfaces.
- All the methods of the interface should be implemented in its sub classes.
- The built-in function globals()[str] converts the string 'str' into a classname and returns the classname.
- All the features (or methods) of the interface should be described in a separate file called 'API documentation file'.
- An abstract class is written when there are some common features shared by all the objects.
- An interface is written when all the features are implemented differently for different objects.
- When an interface is written, any third party vendor can provide sub classes.
- Both the abstract classes and interfaces are examples for polymorphism.