Unit II

Object Oriented programming Concept In Python

Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach. In Python, we can easily create and use classes and objects.

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects.

Major principles of object-oriented programming system are given below.

- Class
- Object
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

```
class ClassName:
        <statement-1>
        .
        .
        <statement-N>
```

Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code.

When we define a class, it needs to create an object to allocate the memory. Consider the following example.

Example:

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year
    def display(self):
        print(self.modelname,self.year)

c1 = car('Toyota', 2016)
c1.display()
```

Output:

```
Toyota 2016
```

In the above example, we have created the class named car, and it has two attributes modelname and year. We have created a c1 object to access the class attribute. The c1 object will allocate memory for these values.

Method

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

Inheritance

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides the re-usability of the code.

Polymorphism

Polymorphism contains two words ‘poly‘ and ‘morphs‘. Poly means many, and morph means shape. By polymorphism, we understand that one task can be performed in different ways. For example - you have a class animal, and all animals speak. But they speak differently. Here, the ‘speak‘ behavior is polymorphic in a sense and depends on the animal. So, the abstract ‘animal‘ concept does not actually ‘speak‘, but specific animals (like dogs and cats) have a concrete implementation of the action ‘speak‘.

Encapsulation

Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

Classes and Objects in Python

Python is an object-oriented programming language that offers classes, which are a potent tool for writing reusable code. To describe objects with shared characteristics and behaviours, classes are utilised. We shall examine Python's ideas of classes and objects in this article.

Classes in Python:

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

Creating Classes in Python

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

Syntax

```
class ClassName:
    #statement_suite
```

In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.__doc__**. A class contains a statement suite including fields, constructor, function, etc. definition.

Example:

Code:

```
class Person:
    def __init__(self, name, age):
        # This is the constructor method that is called when creating a new Person object
        # It takes two parameters, name and age, and initializes them as attributes of the object
        self.name = name
        self.age = age
    def greet(self):
        # This is a method of the Person class that prints a greeting message
        print('Hello, my name is ' + self.name)
```

Name and age are the two properties of the Person class. Additionally, it has a function called greet that prints a greeting.

Objects in Python:

An object is a particular instance of a class with unique characteristics and functions. After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python. The object's attributes are initialised in the constructor, which is a special procedure with the name __init__.

Syntax:

```
# Declare an object of a class
object_name = Class_Name(arguments)
```

Example:

Code:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        print('Hello, my name is ' + self.name)

# Create a new instance of the Person class and assign it to the variable person1
person1 = Person('Ayan', 25)
person1.greet()
```

Output:

"Hello, my name is Ayan"

The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

_ _init_ _ method

In order to make an instance of a class in Python, a specific function called __init__ is called. Although it is used to set the object's attributes, it is often referred to as a constructor.

The self-argument is the only one required by the __init__ method. This argument refers to the newly generated instance of the class. To initialise the values of each attribute associated with the objects, you can declare extra arguments in the __init__ method.

Class and Instance Variables

All instances of a class exchange class variables. They function independently of any class methods and may be accessed through the use of the class name. Here's an illustration:

Code:

```
1. class Person:
2.     count = 0    # This is a class variable
3.
4.     def __init__(self, name, age):
5.         self.name = name     # This is an instance variable
```

```
6.          self.age = age
7.          Person.count += 1    # Accessing the class variable using the nam
    e of the class
8. person1 = Person('Ayan', 25)
9. person2 = Person('Bobby', 30)
10. print(Person.count)
```

Output:

```
2
```

Whereas, instance variables are specific to each instance of a class. They are specified using the self-argument in the __init__ method. Here's an illustration:

Code:

```
1. class Person:
2.     def __init__(self, name, age):
3.         self.name = name     # This is an instance variable
4.         self.age = age
5. person1 = Person('Ayan', 25)
6. person2 = Person('Bobby', 30)
7. print(person1.name)
8. print(person2.age)
```

Output:

```
Ayan
30
```

Class variables are created separately from any class methods and are shared by all class copies. Every instance of a class has its own instance variables, which are specified in the __init__ method utilising the self-argument.

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python

In Python, the method the **__init__**() simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

We can pass any number of arguments at the time of creating the class object, depending upon the **__init__**() definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

Example

```
1. class Employee:
2.     def __init__(self, name, id):
```

```
3.          self.id = id
4.          self.name = name
5.
6.      def display(self):
7.          print('ID: %d \nName: %s' % (self.id, self.name))
8.
9.
10.emp1 = Employee('John', 101)
11.emp2 = Employee('David', 102)
12.
13.# accessing display() method to print employee 1 information
14.
15.emp1.display()
16.
17.# accessing display() method to print employee 2 information
18.emp2.display()
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Counting the number of objects of a class

The constructor is called automatically when we create the object of the class. Consider the following example.

Example

```
1. class Student:
2.     count = 0
3.     def __init__(self):
4.         Student.count = Student.count + 1
5. s1=Student()
6. s2=Student()
7. s3=Student()
8. print('The number of students:',Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

Example

```
1. class Student:
2.     # Constructor - non parameterized
3.     def __init__(self):
4.         print('This is non parametrized constructor')
5.     def show(self,name):
```

```
6.          print('Hello' ,name)
7.  student = Student()
8.  student.show('John' )
```

Output:

```
This is parametrized constructor
Hello John
```

Class Method

The **classmethod()** is an inbuilt function in Python, which returns a class method for a given function.;

Syntax: classmethod(function)

Parameter :This function accepts the function name as a parameter.

Return Type:This function returns the converted class method.

Python classmethod()

The classmethod() methods are bound to a class rather than an object. Class methods can be called by both class and object. These methods can be called with a class or with an object.

Class Method vs Static Method

The basic difference between the class method vs Static method in Python and when to use the class method and static method in Python.

A class method takes cls as the first parameter while a static method needs no specific parameters.

A class method can access or modify the class state while a static method can't access or modify it.

In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.

We use @classmethod decorator in Python to create a class method and we use @staticmethod decorator to create a static method in Python.

Inheritance in Python

One of the core concepts in object-oriented programming (OOP) languages is inheritance. It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. Inheritance is the capability of one class to derive or inherit the properties from another class.

Benefits of inheritance are:

- Inheritance allows you to inherit the properties of a class, i.e., base class to another, i.e., derived class. The benefits of Inheritance in Python are as follows:
- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
- Inheritance offers a simple, understandable model structure.
- Less development and maintenance expenses result from an inheritance.

Python Inheritance Syntax

The syntax of simple inheritance in Python is as follows:

```
Class BaseClass:
    {Body}
Class DerivedClass(BaseClass):
    {Body}
```

Creating a Parent Class

A parent class is a class whose properties are inherited by the child class. Let's create a parent class called **Person** which has a **Display** method to display the person's information.

```
# parent class
class Person():
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def display(self):
    print(self.name, self.age)

# child class
class Student(Person):
  def __init__(self, name, age):
    self.sName = name
    self.sAge = age
    # inheriting the properties of parent class
    super().__init__('Rahul', age)

  def displayInfo(self):
    print(self.sName, self.sAge)

obj = Student('Mayank', 23)
obj.display()
obj.displayInfo()
```

Output:

Rahul 23

Mayank 23

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

The syntax of multi-level inheritance is given below.

Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

Example

```
class Animal:
    def speak(self):
        print('Animal Speaking')
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print('dog barking')
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print('Eating bread...')
```

```
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.

The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

Example

```
1. class Calculation1:
2.     def Summation(self,a,b):
3.         return a+b;
4. class Calculation2:
5.     def Multiplication(self,a,b):
6.         return a*b;
7. class Derived(Calculation1,Calculation2):
8.     def Divide(self,a,b):
9.         return a/b;
10.d = Derived()
11.print(d.Summation(10,20))
12.print(d.Multiplication(10,20))
13.print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

Consider the following example.

The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

```
1. class Calculation1:
2.     def Summation(self,a,b):
3.         return a+b;
4. class Calculation2:
5.     def Multiplication(self,a,b):
6.         return a*b;
7. class Derived(Calculation1,Calculation2):
8.     def Divide(self,a,b):
9.         return a/b;
10.d = Derived()
11.print(issubclass(Derived,Calculation2))
12.print(issubclass(Calculation1,Calculation2))
```

Output:

```
True
False
```

The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

Example

```
1. class Calculation1:
2.     def Summation(self,a,b):
3.         return a+b;
4. class Calculation2:
5.     def Multiplication(self,a,b):
6.         return a*b;
7. class Derived(Calculation1,Calculation2):
8.     def Divide(self,a,b):
9.         return a/b;
10.d = Derived()
11.print(isinstance(d,Derived))
```

Output:

True

Method Overriding

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

Example

```
1. class Animal:
2.     def speak(self):
3.         print('speaking' )
```

```
class Dog(Animal):
    def speak(self):
        print('Barking' )
d = Dog( )
d.speak( )
```

Output:

```
Barking
```

Real Life Example of method overriding

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank( )
b2 = SBI( )
b3 = ICICI( )
print('Bank Rate of interest:' ,b1.getroi( ));
print('SBI Rate of interest:' ,b2.getroi( ));
print('ICICI Rate of interest:' ,b3.getroi( ));
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

Data abstraction in python

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Consider the following example.

Example

```
1. class Employee:
2.     __count = 0;
3.     def __init__(self):
4.         Employee.__count = Employee.__count+1
5.     def display(self):
6.         print('The number of employees',Employee.__count)
7. emp = Employee()
8. emp2 = Employee()
9. try:
10.     print(emp.__count)
11. finally:
12.     emp.display()
```

Output:

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```

Method Overloading:

Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

Like other languages (for example, method overloading in C++) do, python does not support method overloading by default. But there are different ways to achieve method overloading in Python.

The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

Example

```
class Mathematics:
    def add(self, *args):
        sum = 0
        for a in args:
            sum = sum + a
        print(sum)

obj = Mathematics()
obj.add(8, 9, 12)
obj.add(8, 9)
```

Output of the above program

```
29
17
```

Difference between Method Overloading and Overriding

Method Overloading	Method Overriding
Refers to defining multiple methods with the same name but different parameters	Refers to defining a method in a subclass that has the same name as the one in its superclass
Can be achieved in Python using default arguments	Can be achieved by defining a method in a subclass with the same name as the one in its superclass
Allows a class to have multiple methods with the same name but different behaviors based on the input parameters	Allows a subclass to provide its own implementation of a method defined in its superclass
The choice of which method to call is determined at compile-time based on the number and types of arguments passed to the method	The choice of which method to call is determined at runtime based on the actual object being referred to
Not supported natively in Python	Supported natively in Python

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

```
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len('geeks'))

# len() being used for a list
print(len([10, 20, 30]))
```

Output

5

3

PRogramme:

```
class Animal:
    def speak(self):
        raise NotImplementedError('Subclass must implement this method')

class Dog(Animal):
    def speak(self):
        return 'Woof!'

class Cat(Animal):
    def speak(self):
        return 'Meow!'

# Create a list of Animal objects
animals = [Dog(), Cat()]

# Call the speak method on each object
for animal in animals:print(animal.speak())
```