Chapter 9: Object Oriented programming 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 as follows

- 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.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of Vehicles that may have different attributes like name, colour. If a list is used, the first element could be the Vehicle's name while the second element could represent its color. Let's suppose there are 100 different vehicles, then how would you know which element is supposed to be which? What if you wanted to add other properties to these vehicles? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
# Python3 program to
# demonstrate defining
# a class
      class ClassName:
          <statement-1>
          .
          .
          <statement-N>
```

Example: Creating an empty Class in Python

```
class Vehicle:
    pass
```

Objects

The object is an entity that has a state and behaviour associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of :

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behaviour:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behaviour, and identity let us take the example of the class Vehicle.

- The **identity** can be considered as the name of the Vehicle.
- **State** or Attributes can be considered as the name, model, or color of the Vehicle.
- The **behaviour** can be considered as to whether the Vehicle is moving or stopping.

```
class Vehicle:
    color ="black"
    model=2015
    name ="BMW"
    def accelerate(self):
        print("The Vehicle is accelerating",self.color, self.model)
car=Vehicle()
bus=Vehicle()
print(car.color)
car.accelerate()

print(bus.color)
bus.accelerate()
```

In the above example, car and bus are the objects of class Vehicle. When you access the variables and methods using the objects, they give the same values as they are common to both the objects.

A better design would be to have each object have it's own values and this is achieved using constructors.

```
class Vehicle:
    def __init__(self,name,model, color):
        self.name = name
        self.model = model
        self.color=color

    def display(self):
        print(self.name,self.model,self.color)

car = Vehicle("Toyota", 2016,"Black")
car.display()

bike = Vehicle("Splendor", 2020,"Blue")
bike.display()
```

Following are some of the keywords we used in the above code. Let us first see what their meaning and definitions are

The __init__ method

The __init__ method/function is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

In the above example, we have created the class named Vehicle, and it has three attributes name ,model and year. We have created a car object to access the class attributes. The car object will allocate memory for these values.The car object is initialized with it's own values passed on to the __init__ method. Similarly, the bike object has it's own values passed on to it's constructor, the __init__ method

The self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

Python Class and Objects

A class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated.

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.

On the other hand, the object is the instance of a class. The process of creating an object can be called instantiation.

Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

```
<object-name> = <class-name>(<arguments>)
car = Vehicle("Toyota", 2016,"Black")
```

In the above code, we have created the Vehicle class which has three attributes named name, model and color and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **car.** By using it, we can access the attributes of the class.

Delete the Object

We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

```
del car.color
car.display()    # will throw an error as vehicle no longer has the color attribute
```

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 an Employee class

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

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))
```

```
emp1 = Employee("John", 101)
emp2 = Employee("David", 102)

# accessing display() method to print employee 1 information

emp1.display()

# accessing display() method to print employee 2 information
emp2.display()
```

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.

```
class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("John")
```

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects. Consider the following example.

```
class Student:
    roll_num = 101
    name = "Joseph"
    def display(self):
```

```
        print(self.roll_num,self.name)

    st = Student()
    st.display()
```

More than One Constructor in Single class

Let's have a look at another scenario, what happen if we declare the two same constructors in the class.

```
    class Student:
        def __init__(self):
            print("The First Constructor")
        def __init__(self):
            print("The second contructor")

    st = Student()
```

In the above code, the object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the **st** object. Internally, the object of the class will always call the last constructor if the class has multiple constructors.

Some other scenarios for constructors

If the only constructor defined is a parameterized constructor, then you cannot create an object by invoking the default constructor

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age
    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
s = Student()          # Will throw an error
```

The above error is because there is no constructor match the default constructor. This happens only when a parameterized constructor is defined.
To remove this error, there are 2 options. One is to define a new default constructor without any parameters.
Other option is to define default values for the parameterized constructor above as shown below

```
class Student:
    def __init__(self,name='Ravi',id=10,age=30):
        self.name = name;
        self.id = id;
        self.age = age
    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
s = Student()          # Will NOT throw an error
```

Python built-in class functions

The built-in functions defined in the class are described in the following table.

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

```
class Student:
    def __init__(self, name, id, age):
        self.name = name
        self.id = id
        self.age = age
```

```
    # creates the object of the class Student
s = Student("John", 101, 22)

# prints the attribute name of the object s
print(getattr(s, 'name'))

# reset the value of attribute age to 23
setattr(s, "age", 23)

# prints the modified value of age
print(getattr(s, 'age'))

# prints true if the student contains the attribute with name id

print(hasattr(s, 'id'))
# deletes the attribute age
delattr(s, 'age')

# this will give an error since the attribute age has been deleted
print(s.age)
```

Built-in class attributes

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.

4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age
    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
s = Student("John",101,22)
print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

Python | Method Overloading

Like other languages, 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.

```
# First product method.
# Takes two argument and print their
# product
def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
    p = a * b*c
    print(p)

# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

We can achieve method overloading by using Multiple Dispatch Decorator
Multiple Dispatch Decorator Can be installed by:

```
pip3 install multipledispatch
```

Example:

```
from multipledispatch import dispatch

#passing 2 parameters
@dispatch(int,int)
def product(first,second):
    result = first*second
    print(result);

#passing 3 parameters
@dispatch(int,int,int)
def product(first,second,third):
    result  = first * second * third
    print(result);

#you can also pass data type of any value as per requirement
@dispatch(float,float,float)
def product(first, second, third):
    result  = first * second * third
    print(result);

#calling product method with 2 arguments
product(2,3,2) #this will give output of 12
product(2.2,3.4,2.3) # this will give output of 17.985999999999997
```

In Backend, Dispatcher creates an object which stores different implementation and on runtime, it selects the appropriate method as the type and number of parameters passed.

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.

```
class Employee:
   __count = 0;
   def __init__(self):
      Employee.__count = Employee.__count+1
   def display(self):
      print("The number of employees",Employee.__count)
emp = Employee()
emp2 = Employee()
   print(emp.__count)  ## This gives an error, commenting this will give the output
   emp.display()
```

Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

```
class derived-class(base class):
    <class-suite>
```

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
    <class - suite>
```

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("Dog barking")
d = Dog()
d.bark()
d.speak()
```

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.

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

```
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()
```

Python Multiple inheritance

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

```
class Base1:
    <class-suite>

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

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

Example:

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

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.

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

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.

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(isinstance(d,Derived))
```

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.

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

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:Bank = SBI()
b3:Bank = ICICI()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

The __str__ method

In the below example, when we print the object or call a pre-defined method in python__str__(), it prints the default memory location of object v but not the contents of object v. The default __str__() method does not transform the object to a string and instead displays object information in a human-readable format. This is the default behaviour

```
class Vehicle:
  def __init__(self,name,id):
    self.name=name
    self.id=id

v=Vehicle("BMW",10)
print(v)  # OR
print(v.__str__())
```

In order to convert an object to string, override the __str__() method in your own class as shown below

```
class Vehicle:
  def __init__(self,name,id):
    self.name=name
    self.id=id
  def __str__(self):
    return "The name of vehicle is {} and id is {}".format(self.name,self.id)
v=Vehicle("BMW",10)
print(v)
```

Adding 2 objects!!

```
class Data:
    def __init__(self, value):
        self.value = value

a = Data(40)
b = Data(2)
c = a + b       # Throws an error as you cannot add objects
```

#Define the __add__() method

```
class Data:
    def __init__(self, value):
        self.value = value
    def __add__(self,other):
      return Data(self.value + other.value)
    def __str__(self):
      return f"{self.value}"
a=Data(30)
b=Data(20)
print(a+b)
```

Comparing 2 objects

```
class Data:
    def __init__(self, value):
        self.value = value
a=Data(30)
b=Data(20)
a==b      # Gives error
```

Solution: Implement the __eq__ method

```
class Data:
    def __init__(self, value):
        self.value = value
    def __eq__(self,other):
      return self.value == other.value

a=Data(30)
b=Data(20)
a==b     # OK now as we have overridden __eq__() method
```

Exercises

Problem 1: Write a Python class named Circle constructed by a radius and two methods which will compute the area and the perimeter of a circle.

Problem 2: Write a Python class named Rectangle constructed by a length and width and a method which will compute the area of a rectangle.
Problem 3: Write a Python class which has two methods get_String and print_String. get_String accept a string from the user and print_String print the string in upper case.

Problem 4: Write a Python class to reverse a string word by word.

Problem 5: Write a python class called Shape and implement a method area which contains a default implementation. Write 2 subclasses named Rectangle and Circle which has their own implementations of calculation of area. Create objects of both classes and assign them to the same Shape class object type and invoke the area methods