Python OOPs Concepts

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating classes and objects. This is known as Object-Oriented Programming (OOP). 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 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.

Syntax

```
class Animal:
 pass
```

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.

```
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()
```

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. We will learn more about class and object in the next tutorial.

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 behaviours of the parent object.

By using inheritance, we can create a class which uses all the properties and behaviour 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.

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.

Defining a Class in Python

Like function definitions begin with the def keyword in Python, class definitions begin with a class keyword.
The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.
Here is a simple class definition.

```
class MyNewClass:
  '''This is a docstring. I have created a new class'''
  pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.There are also special attributes in it that begins with double underscores __.For example, __doc__ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

Consider the following example to create a class **Employee** which contains two fields as Employee id, and name.

The class also contains a function **display()**, which is used to display the information of the **Employee.**

```
class Employee:
    id = 10
    name = "Devansh"
    def display (self):
        print(self.id,self.name)
```

Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.

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.

Creating an Object in Python

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 following example creates the instance of the class Employee

```
class Employee:
   id = 10
   name = "John"
   def display (self):
      print("ID: %d \nName: %s"%(self.id,self.name))
# Creating a emp instance of Employee class
emp = Employee()
emp.display()
```

we have created the Employee class which has two attributes named id and name 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 **emp.** By using it, we can access the attributes of the class.

Constructors in Python

In Python, the method **__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.

```
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()
```

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")
```

Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

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

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.

Python built-in class functions

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)
```

Output

```
John
23
True
AttributeError: 'Student' object has no attribute '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.

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

5. __name__ :- It is used to access the class name.

Example:

```
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__)
```

---

output

None

{'name': 'John', 'id': 101, 'age': 22}

__main__

Python Inheritance

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.

Syntax

```
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

Syntax

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

Example:

```
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()
```

========

Output:

dog barking

Animal Speaking

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.

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...
```

--- --- -- -- ---

Multiple inheritance

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

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))
```

--- -- --- ---- --- ----

Output:

30

200

0.5

--- -- --- ---- --- ----

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.

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))
```

--- -- --- -- ---

Output

True

--- -- --- -- ---

Using Super keyword in Python

```
# Python example to show that base class members can be accessed # in derived class using super()
class Base(object):
        def __init__(self, x):
                self.x = x

class Derived(Base):
        def __init__(self, x, y):

                ''' In Python 3.x, "super().__init__(name)"
                        also works'''
                super(Derived, self).__init__(x)
                self.y = y

        def printXY(self):
                print(self.x, self.y)

# Driver Code
d = Derived(10, 20)
d.printXY()
```

--- -- --- -- --- -- ---

Output:

(10, 20)

--- -- --- -- --- -- ---

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.

Example:

```
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.

```
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()
```

```
try:
   print(emp.__count)
finally:
   emp.display()
--- -- --- ---- --- ----
```

Output:

The number of employees 2

AttributeError: 'Employee' object has no attribute '__count'

--- -- --- ---- --- ----

Class Method

The @classmethod decorator is a built-in function decorator that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition.

A class method receives the class as an implicit first argument, just like an instance method receives the instance

Syntax:

```
class C(object):
   @classmethod
   def fun(cls, arg1, arg2, ...):
      ....
```

fun: function that needs to be converted into a class method
returns: a class method for function.

1. A class method is a method that is bound to the class and not the object of the class.
2. They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
3. It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances.

Static Method

A static method does not receive an implicit first argument.

Syntax:

```
class C(object):
   @staticmethod
   def fun(arg1, arg2, ...):
      ...
```

returns: a static method for function fun

1. A static method is also a method that is bound to the class and not the object of the class.
2. A static method can't access or modify the class state.
3. It is present in a class because it makes sense for the method to be present in class.

Class method vs Static Method

- 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.

When to use what?

We generally use class method to create factory methods. Factory methods return class objects (similar to a constructor) for different use cases.

We generally use static methods to create utility functions.

Example:

```
# Python program to demonstrate
# use of class method and static method.
from datetime import date

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

# a class method to create a Person object by birth year.
        @classmethod
        def fromBirthYear(cls, name, year):
                return cls(name, date.today().year - year)

# a static method to check if a Person is adult or not.
        @staticmethod
        def isAdult(age):
                return age > 18

person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)
```

```
print (person1.age)
print (person2.age)

# print the result
print (Person.isAdult(22))
```

--- --- -- ---- ---- ---- ---

Output:

```
21
25
True
```

--- --- -- ---- ---- ---- ---