Python OOPs Concepts

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.

Syntax

```
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:

In [1]:
```
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()
```

```
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. 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 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.

Object-oriented vs. Procedure-oriented Programming languages

The difference between object-oriented and procedure-oriented programming is given below:

Object-oriented programming

1. Object-oriented programming is the problem-solving approach and used where computation is done by using objects.
2. It makes the development and maintenance easier.
3. It simulates the real world entity. So real-world problems can be easily solved through oops.
4. It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.
5. Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.

Procedural Programming

1. Procedural programming uses a list of instructions to do computation step by step.
2. In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.

3. It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
4. Procedural language doesn't provide any proper way for data binding, so it is less secure.
5. Example of procedural languages are: C, Fortran, Pascal, VB etc.

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 .**doc**. A class contains a statement suite including fields, constructor, function, etc. definition. 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.

Example

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

In [16]:
```
class Car:
    name = "Honda"
    year = "2021"
    cc = 1200
    model = "city"

    def printDetails(self):
        print("Car Name:",Car.name)
        print("Car year:",Car.year)
        print("Car CC:",Car.cc)
        print("Car Model:",Car.model)
```

In [17]:
```
honda = Car()
honda.printDetails()
```

```
Car Name: Honda
Car year: 2021
Car CC: 1200
Car Model: city
```

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 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>)
```

The following example creates the instance of the class Employee defined in the above example.

In [2]:
```
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()
```

```
ID: 10
Name: John
```

In the above code, 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.

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.

Parameterized Constructor 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

In [24]:

```
class Employee:
    company = 'DataTrained'

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def func_message(self):
        print('Welcome to Python Programming')

emp1 = Employee('Mike', 25, 'Male')

print(emp1.company)
emp1.func_message()
print(emp1.name)
print(emp1.age)
print(emp1.gender)
```

```
DataTrained
Welcome to Python Programming
Mike
25
Male
```

In [25]:
```
#Multiple objects
emp2 = Employee('Tracy', 27, 'Female')
print(emp2.company)
emp2.func_message()
print(emp2.name)
print(emp2.age)
print(emp2.gender)
```

```
DataTrained
Welcome to Python Programming
Tracy
27
Female
```

In [26]:
```
class Circle(object):
    pi = 3.14159

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return Circle.pi * self.radius * self.radius
```

In [27]:
```

Circle.pi
## 3.14159

c = Circle(10)
c.pi
## 3.14159
c.area()
## 314.159
```

Out[27]:
```
314.159
```

In [31]:
```
class Vehicle:
    def __init__(self, brand, model, type):
        self.brand = brand
        self.model = model
        self.type = type
        self.gas_tank_size = 14
        self.fuel_level = 0

    def fuel_up(self):
        self.fuel_level = self.gas_tank_size
        print('Gas tank is now full.')

    def drive(self):
        print(f'The {self.model} is now driving.')
```

In [32]:
```
vehicle_object = Vehicle('Honda', 'Ridgeline', 'Truck')


#Accessing attribute values
print(vehicle_object.brand)
print(vehicle_object.model)
print(vehicle_object.type)
```

```
Honda
Ridgeline
Truck
```

In [33]:
```
#Calling methods


vehicle_object.fuel_up()
vehicle_object.drive()

```

```
Gas tank is now full.
The Ridgeline is now driving.
```

In [34]:
```
#Creating multiple objects

vehicle_object = Vehicle('Honda', 'Ridgeline', 'Truck')
a_subaru = Vehicle('Subaru', 'Forester', 'Crossover')
an_suv = Vehicle('Ford', 'Explorer', 'SUV')
```

In [4]:
```
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()
```

```
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

In [5]:
```
class Student:
    count = 0
    def __init__(self):
        Student.count = Student.count + 1
s1=Student()
s2=Student()
s3=Student()
print("The number of students:",Student.count)

```

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

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 of the tutorial, 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.

Syntax

```

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

In [20]:
```
class Car1:
    carname = "Honda"

# Class 2 inherting the properties of class 1
class Car2(Car1):
    carModel = "City"
car2Instance = Car2()
```

In [23]:

```
# calling the attribute of class 2
print(car2Instance.carname)

# calling the attribute of class 1
print(car2Instance.carModel)
```

```
Honda
City
```

In [ ]:

```

```