What Is Object-Oriented Programming?

Object-Oriented Programming(OOP), is all about creating “objects”. An object is a group of interrelated variables and functions. These variables are often referred to as properties of the object and functions are referred to as the behavior of the objects. These objects provide a better and clear structure for the program.

For example, a car can be an object. If we consider the car as an object then its properties would be – its color, its model, its price, its brand, etc. And its behavior/function would be acceleration, slowing down, gear change.

What is a Class?

A straight forward answer to this question is- A class is a collection of objects. Unlike the primitive data structures, classes are data structures that the user defines. They make the code more manageable.

OR

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Consider the case of a car showroom. You want to store the details of each car. Let’s start by defining a class first-

In [1]:

```
class Car:
    pass
```

When we define a class only the description or a blueprint of the object is created. There is no memory allocation until we create its object. The objector instance contains real data or information.

Instantiation is nothing but creating a new object/instance of a class. Let’s create the object of the above class we defined-

In [2]:

```
#Let’s create the object of the above class we defined-

obj1 = Car()
```

In [3]:

```
#Try printing this object-

print(obj1)
```

```
<__main__.Car object at 0x0000028999863D30>
```

Since our class was empty, it returns the address where the object is stored i.e 0x7fc5e677b6d8

Class Constructor

Until now we have an empty class Car, time to fill up our class with the properties of the car. The job of the class constructor is to assign the values to the data members of the class when an object of the class is created.

There can be various properties of a car such as its name, color, model, brand name, engine power, weight, price, etc. We'll choose only a few for understanding purposes.

In [ ]:

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

So, the properties of the car or any other object must be inside a method that we call **init**( ). This **init**() method is also known as the constructor method. We call a constructor method whenever an object of the class is constructed.

Now let's talk about the parameter of the **init**() method. So, the first parameter of this method has to be self. Then only will the rest of the parameters come.

Note: You can create attributes outside of this **init**() method also. But those attributes will be universal to the whole class and you will have to assign the value to them.

Suppose all the cars in your showroom are Sedan and instead of specifying it again and again you can fix the value of car_type as Sedan by creating an attribute outside the **init**().

In [4]:

```
class Car:
    car_type = "Sedan"                  #class attribute
    def __init__(self, name, color):
      self.name = name                 #instance attribute
      self.color = color               #instance attribute
```

Class methods

So far we've added the properties of the car. Now it's time to add some behavior. Methods are the functions that we use to describe the behavior of the objects. They are also defined inside a class.

Look at the following code-

In [10]:

```
class Car:
    car_type = "suv"

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"

    def max_speed(self, speed):
        return f"The {self.name} runs at the maximum speed of {speed}km/hr"
```

The methods defined inside a class other than the constructor method are known as the instance methods.

Furthermore, we have two instance methods here- description() and max_speed().

Notice that the additional parameter speed is not using the “self” keyword. Since speed is not an instance variable, we don’t use the self keyword as its prefix.

Let’s create an object for the class described above.

In [11]:

```
obj2 = Car("Honda City",24.1)
print(obj2.description())
print(obj2.max_speed(150))
```

```
The Honda City car gives the mileage of 24.1km/l
The Honda City runs at the maximum speed of 150km/hr
```

The method description() didn’t have any additional parameter so we did not pass any argument while calling it.

The method max_speed() has one additional parameter so we passed one argument while calling it.

Note: Three important things to remember are-

1. You can create any number of objects of a class.
2. If the method requires n parameters and you do not pass the same number of arguments then an error will occur.
3. Order of the arguments matters.

Creating more than one object of a class

In [22]:

```
class Car:
    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def max_speed(self, speed):
        return f"The {self.name} runs at the maximum speed of {speed}km/hr"

Honda = Car("Honda city",21)
print(Honda.max_speed(150))

Skoda = Car("Skoda Octavia",13)
print(Skoda.max_speed(210))
```

```
The Honda city runs at the maximum speed of 150km/hr
The Skoda Octavia runs at the maximum speed of 210km/hr
```

In [33]:

```
class bird:
    def __init__(self,name,colour):
        self.name=name
        self.colour=colour
    def Bird_size(self,size):
        return f"The {self.name} is {self.colour} in colour and size is {size}"

birds= bird('Parrot','green')
print(birds.Bird_size("medium"))
```

```
The Parrot is green in colour and size is medium
```

Encapsulation

Encapsulation, as I mentioned in the initial part of the article, is a way to ensure security. Basically, it hides the data from the access of outsiders. Such as if an organization wants to protect an object/information from unwanted access by clients or any unauthorized person then encapsulation is the way to ensure this.

You can declare the methods or the attributes protected by using a single underscore (_) before their names. Such as- self._name or def _method( ); Both of these lines tell that the attribute and method are protected and should not be used outside the access of the class and sub-classes but can be accessed by class methods and objects.

Now for actually preventing the access of attributes/methods from outside the scope of a class, you can use “private members“. In order to declare the attributes/method as private members, use double underscore (__) in the prefix. Such as – self.__name or def __method(); Both of these lines tell that the attribute and method are private and access is not possible from outside the class.

In [ ]:

```
class car:

    def __init__(self, name, mileage):
        self._name = name                #protected variable
        self.mileage = mileage

    def description(self):
        return f"The {self._name} car gives the mileage of {self.mileage}km/l"
obj = car("BMW 7-series",39.53)

#accessing protected variable via class method
print(obj.description())

#accessing protected variable directly from outside
print(obj._name)
print(obj.mileage)
```

Notice how we accessed the protected variable without any error. It is clear that access to the variable is still public. Let us see how encapsulation works-

In [ ]:

```
class Car:

    def __init__(self, name, mileage):
        self.__name = name              #private variable
        self.mileage = mileage

    def description(self):
        return f"The {self.__name} car gives the mileage of {self.mileage}km/l"
obj = Car("BMW 7-series",39.53)

#accessing private variable via class method
print(obj.description())

#accessing private variable directly from outside
print(obj.mileage)
print(obj.__name)
```

When we tried accessing the private variable using the description() method, we encountered no error. But when we tried accessing the private variable directly outside the class, then Python gave us an error stating: car object has no attribute '__name'.

You can still access this attribute directly using its mangled name. Name mangling is a mechanism we use for accessing the class members from outside.

In [ ]:

```
class Car:

    def __init__(self, name, mileage):
        self.__name = name              #private variable
        self.mileage = mileage

    def description(self):
        return f"The {self.__name} car gives the mileage of {self.mileage}km/l"
obj = Car("BMW 7-series",39.53)

#accessing private variable via class method
print(obj.description())

#accessing private variable directly from outside
print(obj.mileage)
print(obj._Car__name)      #mangled name
```

Inheritance in Python Class

Inheritance is the procedure in which one class inherits the attributes and methods of another class. The class whose properties and methods are inherited is known as Parent class. And the class that inherits the properties from the parent class is the Child class.

The interesting thing is, along with the inherited properties and methods, a child class can have its own properties and methods.

In [ ]:

```
class Car:          #parent class

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"

class BMW(Car):     #child class
    pass

class Audi(Car):     #child class
    def audi_desc(self):
        return "This is the description method of class Audi."
obj1 = BMW("BMW 7-series",39.53)
print(obj1.description())

obj2 = Audi("Audi A8 L",14)
print(obj2.description())
print(obj2.audi_desc())
```

We have created two child classes namely “BMW” and “Audi” who have inherited the methods and properties of the parent class “Car”. We have provided no additional features and methods in the class BMW. Whereas one

additional method inside the class Audi.

Notice how the instance method description() of the parent class is accessible by the objects of child classes with the help of obj1.description() and obj2.description(). And also the separate method of class Audi is also accessible using obj2.audi_desc().

In [ ]:

```
#Example
 # A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Nikhil')
p.say_hi()
```

In [ ]:

```
Exercise :
# Python3 program to show that the variables with a value
# assigned in the class declaration, are class variables and
# variables inside methods and constructors are instance
# variables.

# Class for Dog
class Dog:

    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed, color):

        # Instance Variable
        self.breed = breed
        self.color = color

# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using class
# name also
print("\nAccessing class variable using class name")
print(Dog.animal)
```

Method Overloading

1. In the method overloading, methods or functions must have the same name and different signatures.
2. Method overloading is a example of compile time polymorphism.
3. In the method overloading, inheritance may or may not be required.
4. Method overloading is performed between methods within the class.
5. It is used in order to add more to the behavior of methods.
6. In method overloading, there is no need of more than one class.

In [ ]:

```
class area:
  def find_area(self, x = None, y = None):

    if x != None and y != None:
        print(x*y)
    elif x != None:
        print(x*x)
    else:
        print("Nothing")

obj1 = area()
obj1.find_area()
obj1.find_area(10)
obj1.find_area(10,20)
```

Method Overriding

1. Whereas in the method overriding, methods or functions must have the same name and same signatures.
2. Whereas method overriding is a example of run time polymorphism.
3. Whereas in method overriding, inheritance always required.
4. Whereas method overriding is done between parent class and child class methods.
5. Whereas it is used in order to change the behavior of exist methods.
6. Whereas in method overriding, there is need of at least of two classes.

In [ ]:

```
class A:
  def Data(self):
    print("I'm in class A")
class B(A):
  def Data(self):
    print("I'm in class B")

obj = B()
obj.Data()
```

In [ ]:

```

```