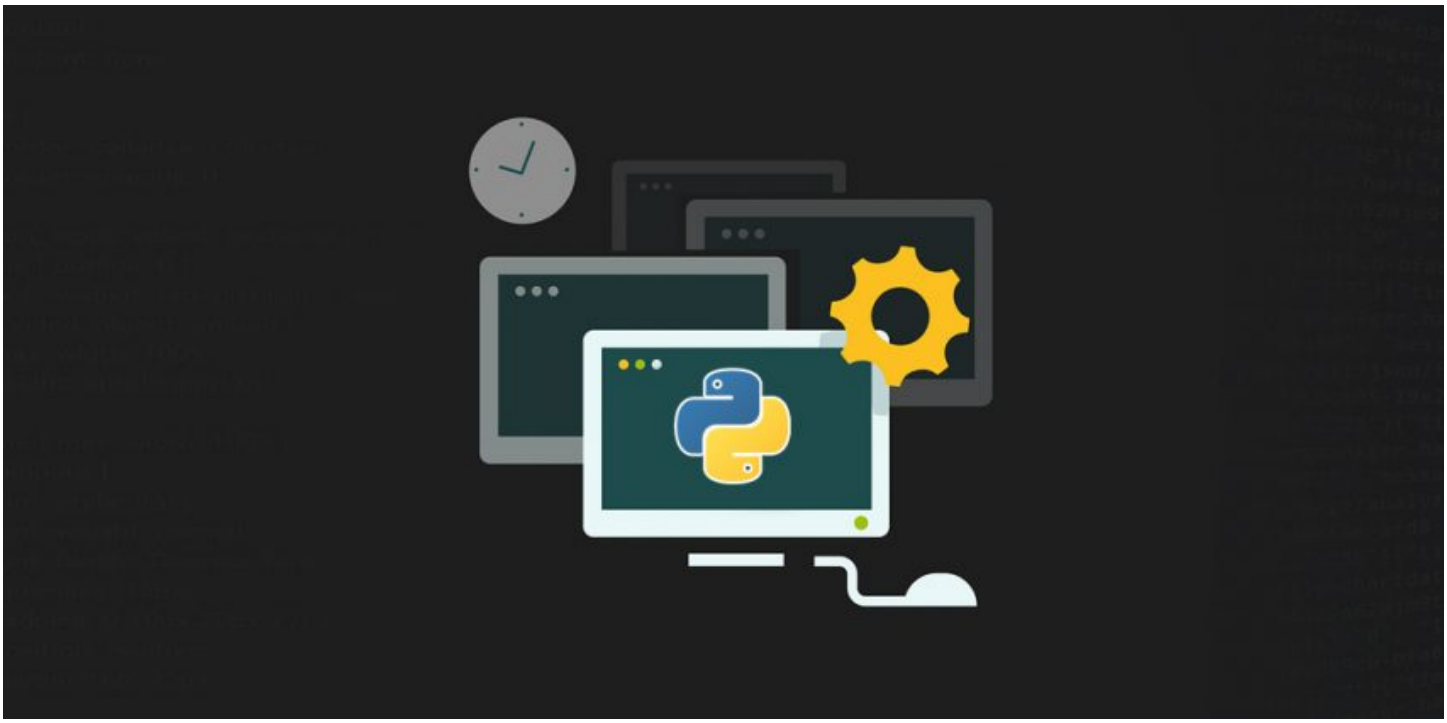
Basic Concepts of Object-Oriented Programming in Python

BEGINNER PROGRAMMING PYTHON TECHNIQUE

Introduction

While learning Object-Oriented Programming. I decided to dive into its history and it turned out to be fascinating. The term “Object-Oriented Programming” (OOP) was coined by Alan Kay around 1966 while he was at grad school. The language called **Simula** was the first programming language with the features of Object-oriented programming. It was developed in 1967 for making simulation programs, in which the most important information was called objects.



Though OOPs were in the market since the early 1960s it was in the 1990s that OOPs began to grow because of C++. After that, this technique of programming has been adapted by various programming languages including Python Today its application is in almost every field such as Real-time systems, Artificial intelligence, and expert systems, Client-server systems, Object-oriented databases, and many more.

So, in this article, I will explain the basic concepts of Object-Oriented Programming in Python. It is important that you know Python before you continue. You can learn Python using the free course mentioned below-

- *Introduction to Python*

Table of Content

1. What Is Object-Oriented Programming?
2. Object-Oriented Programming (OOP) vs Procedure Oriented Programming (POP)
3. Major Concepts of OOPs-
    1. What is a class?
    2. Object and object instantiation
    3. Class methods
    4. Inheritance in Python Class
    5. Encapsulation
    6. Polymorphism
    7. Data abstraction

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.

Another example- If we consider a dog as an object then its properties would be- his color, his breed, his name, his weight, etc. And his behavior/function would be walking, barking, playing, etc.

Object-Oriented programming is famous because it implements the read world entities like objects, hiding, inheritance, etc in programming. It makes visualization easier because it is close to real-world scenarios.

Object-Oriented Programming (OOP) vs Procedure Oriented Programming (POP)

The basic difference between OOP and procedural programming is-

- One way to think about POP is the same way you make lemonade for example. The procedure of making lemonade involves- first taking water according to the need, then adding sugar to the water, then adding lemon juice to the mixture, and finally mixing the whole solution. And your lemonade is ready to serve. In a similar way POP requires a certain procedure of steps. A procedural program consists of functions. This means that in the POP approach the program is divided into functions, which are specific to different tasks. These functions are arranged in a specific sequence and the control of the program flows sequentially.

  Whereas an OOP program consists of objects. The object-Oriented approach divides the program into objects. And these objects are the entities that bundle up the properties and the behavior of the real-world objects.

- POP is suitable for small tasks only. Because as the length of the program increases, the complexity of the code also increases. And it ends up becoming a web of functions. Also, it becomes hard to debug. OOP solves this problem with the help of a clearer and less complex structure. It allows code re-usability in the form of inheritance.

- Another important thing is that in procedure-oriented programming all the functions have access to all the data, which implies a lack of security. Suppose you want to secure the credentials or any other critical information from the world. Then the procedural approach fails to provide you that security. For this OOP helps you with one of its amazing functionality known as **Encapsulation**, which allows us to hide data. Don't worry I'll cover this in detail in the latter part of this article along with other concepts of Object-Oriented Programming. For now, just understand that OOP enables security and POP does not.

- Programming languages like C, Pascal and BASIC use the procedural approach whereas Java, Python, JavaScript, PHP, Scala, and C++ are the main languages that provide the Object-oriented approach.

Major Python OOPs concept-

In this section, we will deep dive into the basic concepts of OOP. We will cover the following topics-

1. Class
2. Object
3. Method
4. Inheritance
5. Encapsulation
6. Polymorphism
7. Data Abstraction

1. 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.

Let's see how to define a class below-

```
class class_name: class body
```

We define a class with a keyword "class" following the class_name and semicolon. And we consider everything you write under this after using indentation as its body. To make this more understandable let's see an example.

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

```
class Car: pass
```

That's it!

Note: I've used the pass statement in place of its body because the main aim was to show how you can define a class and not what it should contain.

Before going in detail, first, understand objects and instantiation.

2. Objects and object instantiation

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-

```
obj1 = Car()
```

And it's done! **Note** that you can change the object name according to your choice.

Try printing this object-

```
print(obj1)
```

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

You also need to understand class conductor before moving forward.

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.

```
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.

The two statements inside the constructor method are –

1. **self.name = name**
2. **self.color = color:**

This will create new attributes namely **name** and **color** and then assign the value of the respective parameters to them. The “self” keyword represents the instance of the class. By using the “self” keyword we can access the attributes and methods of the class. It is useful in method definitions and in variable initialization. The “self” is explicitly used every time we define a method.

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__().

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

Here, **Instance attributes refer to** the attributes inside the constructor method i.e self.name and self.color. And, **Class attributes refer to** the attributes outside the constructor method i.e car_type.

3. 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-

```
class Car: car_type = "Sedan" 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**(). Let’s talk about them individually-

- **description**()- This method is returning a string with the description of the car such as the name and its mileage. This method has no additional parameter. This method is using the instance attributes.
- **max_speed**()- This method has one additional parameter and returning a string displaying the car name and its 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.

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

What we did is we created an object of class car and passed the required arguments. In order to access the instance methods we use object_name.method_name().

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.

Let's look at this one by one-

1. **Creating more than one object of a class**

```
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",21.4)   print(Honda.max_speed(150))   Skoda   =   Car("Skoda   Octavia",13)
print(Skoda.max_speed(210))
```

2. **Passing the wrong number of arguments.**

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


Honda = Car("Honda City") print(Honda)
```

   Since we did not provide the second argument, we got this error.

3. **Order of the arguments**

```
class Car:    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"


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

Messed up! Notice we changed the order of arguments.

Now, there are four fundamental concepts of Object-oriented programming – **Inheritance**, **Encapsulation**, **Polymorphism**, and **Data abstraction**. It is very important to know about all of these in order to understand OOPs. Till now we've covered the basics of OOPs, let's dive in further.

4. 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.

How to inherit a parent class? Use the following syntax:

```
class parent_class: body of parent class class child_class( parent_class): body of child class
```

Let's see the implementation-

```
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().

5. 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.

Though Python uses ‘ _ ‘ just as a coding convention, it tells that you should use these attributes/methods within the scope of the class. But you can still access the variables and methods which are defined as protected, as usual.

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.

```
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-

```
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. The Python interpreter rewrites any identifier with "__var" as "_ClassName__var". And using this you can access the class member from outside as well.

```
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
```

Note that the mangling rule's design mostly avoids accidents. But it is still possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

6. Polymorphism

This is a Greek word. If we break the term Polymorphism, we get "poly"-many and "morph"-forms. So Polymorphism means having many forms. In OOP it refers to the functions having the same names but carrying different functionalities.

```
class Audi:   def description(self):       print("This the description function of class AUDI.") class BMW:
 def description(self):     print("This the description function of class BMW.")



audi = Audi() bmw = BMW() for car in (audi,bmw): car.description()
```

When the function is called using the object *audi* then the function of class *Audi* is called and when it is called using the object *bmw* then the function of class *BMW* is called.

7. Data abstraction

We use Abstraction for hiding the internal details or implementations of a function and showing its functionalities only. This is similar to the way you know how to drive a car without knowing the background mechanism. Or you know how to turn on or off a light using a switch but you don't know what is happening behind the socket.

Any class with at least one abstract function is an abstract class. In order to create an abstraction class first, you need to import ABC class from abc module. This lets you create abstract methods inside it. ABC stands for Abstract Base Class.

```
from abc import ABC class abs_class(ABC):     Body of the class
```

Important thing is– you cannot create an object for the abstract class with the abstract method. For example-

```
from abc import ABC, abstractmethod class Car(ABC): def __init__(self,name):         self.name = name
@abstractmethod def price(self,x): pass


obj = Car("Honda City")
```

Now the question is how do we use this abstraction exactly. The answer is by using inheritance.

```
from abc import ABC, abstractmethod class Car(ABC): def __init__(self,name): self.name = name     def
description(self):          print("This the description function of class car.") @abstractmethod     def
price(self,x): pass class new(Car):  def price(self,x):     print(f"The {self.name}'s price is {x} lakhs.")


obj = new("Honda City") obj.description() obj.price(25)
```

Car is the abstract class that inherits from the **ABC** class from the abc module. Notice how I have an abstract method (price()) and a concrete method (description()) in the abstract class. This is because the abstract class can include both of these kinds of functions but a normal class cannot. The other class inheriting from this abstract class is *new()*. This method is giving definition to the abstract method (price()) which is how we use abstract functions.

After the user creates objects from *new()* class and invokes the price() method, the definitions for the price method inside the *new()* class comes into play. These definitions are hidden from the user. The Abstract method is just providing a declaration. The child classes need to provide the definition.

But when the description() method is called for the object of new() class i.e **obj**, the Car's description() method is invoked since it is not an abstract method.

End Notes

To conclude, in this article I covered the basic concepts of Object-Oriented Programming in Python. Now you guys must have a sense about what OOPs exactly is and its components.

Here are some articles you can read to know more about Python-

- Python Style Guide | How to Write Neat and Impressive Python Code
- Everything you Should Know About Data Structures in Python

I hope you have understood the concepts explained in these articles. Let me know in the comments below if you have any queries.

Article Url - https://www.analyticsvidhya.com/blog/2020/08/object-oriented-programming/

Himanshi Singh

I am a data lover and I love to extract and understand the hidden patterns in the data. I want to learn and grow in the field of Machine Learning and Data Science.