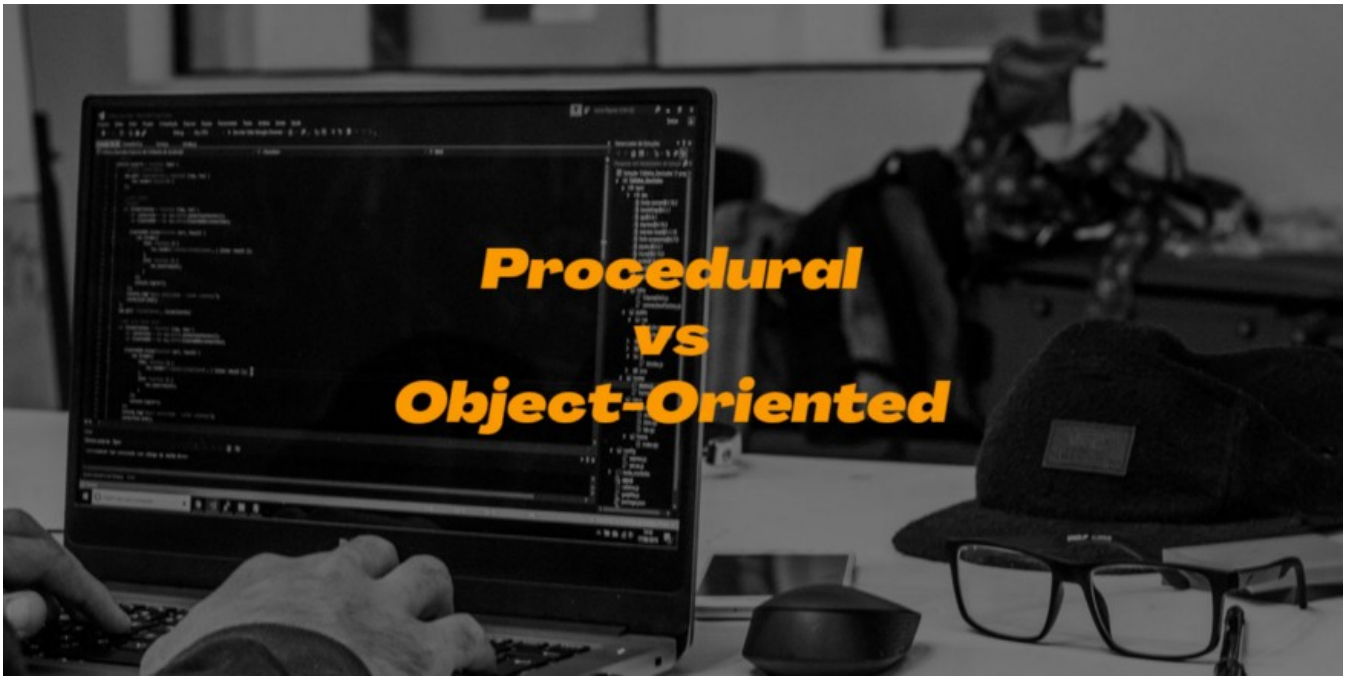

Python: Is it Procedural or Object-Oriented Programming? Photo by David Rangel on Unsplash

DATA SCIENCE AND MACHINE LEARNING

Python: Procedural or Object-Oriented Programming?

It is a little bit argue between procedural and object-oriented, and why we should care about it?

Wie Kiang H.
Jul 31 · 9 min read ★

> *Who does not know Python?*
> *Mostly used in Data Science and Machine Learning.*
> *Let us discuss more about it!*

When you first learn a program, you are seemingly using a technique called **procedural programming**. A procedural program is typically a list of instructions that

execute one after the other starting from the top of the line. On the other hand, **object-oriented programs** are built around well objects. You can think about objects as something that exists in the real world. For instance, if you were going to establish a shoe store, the store itself would be an object. The items in the store, for example, boots and sandals, would also be objects. The cash register would be an object, and even a salesperson would be an object.

Object-oriented programming has several advantages over procedural programming, which is the programming style you most likely first studied.

- Object-oriented programming enables you to develop large, modular programs that can instantly expand over time.
- Object-oriented programs hide the implementation from the end-user.

> Object-oriented program would focus on the specific characteristics of each object and what each object can do.

An object has two fundamental parts, **characteristics, and actions**. For example, several characteristics of a salesperson would include the person's name, address, phone number, and hourly payout, also what a salesperson can do. That might involve selling an item or taking items from a warehouse. As another example, shoes' characteristics could be the color, size, style, and price. What actions could a shoe take? A shoe is an inanimate object, but it could, for example, change its price.

```
In terms of English grammar:

A characteristic would be a noun.
An action would be a verb.

Real-world example: a dog. Some of the characteristics could be:
* weight
* colour
* breed
* height -> These are all nouns.

What actions would a dog take?
* bark
* run
```

```
* bite
* eat -> These are all verbs.
```

An object has characteristics and actions. Characteristics have a specific name. They are called **attributes**, and the actions are called **methods**. The color, size, style, and the price would be called attributes in the shoe example. The action of changing the price would be a method. There are two other terms, which are **object and class**.

An object would be a specific shoe, for instance, a brown shoe with a US size 7.5, and sneaker style with a price of $110 in the shoe example. This brown shoe could change its price. Another object would be something like, a white color shoe with a US size 4.5, and a flip-flop shoe that costs $80. This white shoe could change its price as well.

Do you notice anything special about the brown shoe and white shoe? They both have the same attributes. In other words, they both have a color, size, style, and price. They also have the same method. It is like they came from a blueprint, a generic shoe consisting of all the attributes and methods. This generic version of an object is called a class.

You only need to classify this blueprint known as class one time, and then you can create specific objects from the class over and over again. In other words, you can use the shoe blueprint, to make as many shoe objects as you want, in any size, shape, color, style, and price. **These are fundamental terms in object-oriented programming.**

```
Object-Oriented Programming (OOP) Vocabulary

Class
a blueprint which is consisting of methods and attributes.

Object
an instance of a class. It can help to think of objects as something
in the real world like a yellow pencil, a small dog, a yellow shoe,
etc. However, objects can be more abstract.

Attribute
a descriptor or characteristic. Examples would be colour, length,
size, etc. These attributes can take on specific values like blue, 3
inches, large, etc.

Method
an action that a class or object could take.
```

. . .

The Syntax

If you have not worked with classes before, the syntax might be complicated.

```
[49]
class Shoe:

    def __init__(self, shoe_color, shoe_size, shoe_style, shoe_price):
        self.color = shoe_color
        self.size = shoe_size
        self.style = shoe_style
        self.price = shoe_price

    def change_price(self, new_price):

        self.price = new_price

    def discount(self, discount):

        return self.price * (1 - discount)
```

Figure 1. Shoe class

In Figure 1, it is written a Shoe class. The class has color, size, style, and price attributes. It also has a method to change the price, as well as a method to output a discounted price. Inside the Shoe class, some codes come to shows examples of how to instantiate shoe objects so that you can see how to use a class in a program. Remember that a class represents a blueprint. Hence we are setting up the color, size, style, and price of a generic shoe.

Two things about this might seem a bit odd:

* __init__ ---> Python built-in function
* self -------> variable

Python use **`__init__`** to create a specific shoe object. On the other hand, the **`self`** variable can be tricky to understand in the beginning. **`self`** saves attributes like color, size, and so on, making those attributes available throughout in the Shoe class. **`self`** is essentially a dictionary that holds all of your attributes and the attribute values — checkout the change price method to see how the **`self`** works.

We can pass in `self` as the first method input to have access to the price. The `self` has the price stored inside it, as well as the other attributes like color, size, and style. `self` will always be the first input to your methods if you want to access the attributes.

Note as well that the change price and the discount methods are like general Python functions. For instance, Python functions do not have to return anything, so like in the change price method, it does not return anything. It only changes the value of the price attribute. The discount method, however, does return something it returns the discounted price.

Additional: Function vs Method

A function and method seem very similar; both of them use the same keyword. They also have inputs and return outputs. The contrast is that a ***method is inside a class***, whereas a ***function is outside of a class.***

. . .

Going Deep into the 'self' Variable

If you define two objects, how does Python differentiate between two objects? That is where `self` comes into play. If you call the `change_price` method on `shoe_one`, how does Python know to change the price of `shoe_one` and not of `shoe_two`?

```
def change_price(self, new_price):
    self.price = new_price

shoe_one = Shoe('brown', '7.5', 'sneaker', 110)
shoe_two = Shoe('white', '4.5', 'flip-flop', 80)

shoe_one.change_price(125)
```

`self` tells Python where to look in the computer's memory for the `shoe_one` object, and then Python changes the price of the `shoe_one` object. When you call the `change_price` method, `shoe_one.change_price(125)`, `self` is implicitly passed in.

The word `self` is just a convention. You could actually use any other name as long as you are consistent; however, you should always use `self` rather than some other word, or else you might confuse people.

. . .

Some Notes About Object-Oriented Programming

Now, we are going to write a separate Python script that uses the shoe class code. In the same folder, you need to create another file called **project.py**. Inside this file, we want to use the shoe class. First, you need to import the shoe class by typing `from shoe import Shoe`. The lowercase shoe refers to the **shoe.py** file, while the uppercase shoe refers to the class defined inside this file.

```python
class Shoe:

    def __init__(self, shoe_color, shoe_size, shoe_style, shoe_price):
        self.color = shoe_color
        self.size = shoe_size
        self.style = shoe_style
        self.price = shoe_price

    def change_price(self, new_price):
        self.price = 0.81 * new_price

    def discount(self, discount):
        return self.price * (1 - discount)
```

shoe.py hosted with ♡ by GitHub view raw

shoe.py file

```python
from shoe import Shoe

shoe_one = Shoe('brown', 7.5, 'sneaker', 110)
shoe_two = Shoe('white', 4.5, 'flip-flop', 80)

print(shoe_one.color)
print(shoe_one.price)

shoe_two.change_price(90)
print(shoe_two.price)

shoe one.color = 'blue'
```

```
13  shoe_one.size = 5.0
14  shoe_one.price = 78
```

project.py hosted with ♡ by GitHub view raw

project.py file

You could specify the file and class anything you wanted to. They do not have to be the same. We just did this for convenience. Now, in the **project.py**, you can use the shoe class. If you noticed, the code is now modularized. You will write some code that uses the shoe class.

The shoe class has a method to change the price of the shoe. In **project.py** file, you can see in line 9, `shoe_two.change_price(90)` . In Python, you can also change the values of an attribute with the following syntax, `shoe_one.color = 'blue'` , `shoe_one.size = 5.0` , and `shoe_one.price = 78` . There are some drawbacks to accessing attributes directly versus writing a method for accessing and displaying attributes.

```
In terms of object-oriented programming,
Python's rules are a bit looser than in other programming languages.

In some languages like C++, you can explicitly state whether an
object should be permitted to change or access an attribute value
directly. Python does not have this option.
```

Why might it be better to change the value with a method instead of changing a value directly?

Changing values through a method gives you more flexibility in the long-term. But if the units of measurement replaced, for example, the store was initially meant to serve in US dollars and now has to work in euros. If you have changed an attribute directly like in line 14, if all of a sudden you have to use euros, you are going to have to modify this manually. You are going to have to do this wherever you access the price attribute directly.

If, on the other hand, you would use a method like the change price method, then all you have to do is go into the shoe class and change the original method one time. We are going to multiply by, for example, 0.81 to convert everything from dollars to euros.

If you go back to **project.py**, a line like a number 9, you do not have to manually change the price from dollars to euros because the conversion will take care of that for you in the shoe class.

. . .

Inheritance

There is one more object-oriented topic to cover in this article, inheritance. We think inheritance is easier to understand using a real-world object like the shoe example from earlier. The shoe had four attributes, color, size, style, and price. The shoe also had two methods: a method to change the price, and a method for calculating a discounted price.

As the store expands, it might stock other types of shoes like heels, ankle boot, and mules. These footwears have a few attributes and methods in common with the shoe object. They probably all have a color, size, style, and price and, they could all use functions to change the price and calculate a discount.

Why code separate classes for each new footwears when they all have so much in common? Alternatively, you could code apparent shoe class, and then the heels, ankle boot, and mules class could inherit the attributes and methods of the shoe class.

```
1  class Shoe:
2
3      def __init__(self, shoe_color, shoe_size, shoe_style, shoe_price):
4          self.color = shoe_color
5          self.size = shoe_size
6          self.style = shoe_style
7          self.price = shoe_price
8
9      def change_price(self, new_price):
10         self.price = 0.81 * new_price
11
12     def discount(self, discount):
13         return self.price * (1 - discount)
14
15
16 class Heels:
17
18     def __init__(self, shoe_color, shoe_size, shoe_style, shoe_price, shoe_heels_height):
19
```

```
        Shoe.__init__(self, shoe_color, shoe_size, shoe_style, shoe_price)
        self.heels_height = shoe_heels_height

    def double_price(self):
        self.price = 2 * self.price


class Ankle_Boot:

    def __init__(self, shoe_color, shoe_size, shoe_style, shoe_price, shoe_boot_model):

        Shoe.__init__(self, shoe_color, shoe_size, shoe_style, shoe_price)
        self.boot_model = shoe_boot_model

    def calculate_discount(self, discount):
        return self.price * (1 - discount / 2)
```

This looks like a family tree where the shoe is the parent, and heels, ankle boot, and mules, are the children. One benefit is that, as you add more shoe types like ballerinas, you can easily add a new class inheriting from the shoe class. What if you wanted to add a new attribute like material to represent if the footwear is made of synthetic, rubber, or foam? Now, all you have to do is, add a material attribute to the shoe class, and all the children's classes automatically inherit the new attribute. Your code becomes more efficient to write and maintain.

. . .

Conclusion

> Python is considered as an object-oriented programming language rather than a procedural programming language.

It is identified by looking at Python packages like **Scikit-learn**[1], **pandas**[2], and **NumPy**[3]. These are all Python packages built with object-oriented programming. Scikit-learn, for example, is a relatively comprehensive and complicated package built with object-oriented programming. This package has grown over the years with the latest functionality and new algorithms.

When you train a machine learning algorithm with Scikit-learn, you do not have to know anything about how the algorithms work or how they were coded. You can straightly focus on the modeling.

Thus far, we have been exposed to the core of object-oriented programming languages in Python:

- classes and objects
- attributes and methods
- inheritance

Knowing these topics is enough for you to start writing object-oriented software. However, these are only the fundamentals of object-oriented programming. **You can learn more about this by accessing the advanced topics below.** Additionally, the source code from this article is available on my **GitHub**[4].

References

#1 Scikit-learn
#2 pandas
#3 NumPy
#4 Source code on GitHub

Advanced Python Object-Oriented Programming Topics

```
* class methods, instance methods, and static methods
* class attributes vs instance attributes
* multiple inheritance, mixins
* Python decorators
```

. . .

Writer Note: I would love to receive critiques, comments, and suggestions.