Théo Vanderheyden
March 28th, 2018

MUST READ TUTORIAL +1

Python Object-Oriented Programming (OOP): Tutorial

Tackle the basics of Object-Oriented Programming (OOP) in Python: explore classes, objects, instance methods, attributes and much more!

Object-Oriented programming

is a widely used concept to write powerful applications. As a data scientist, you will be required to write applications to process your data, among a range of other things. In this tutorial, you will discover the basics of object-oriented programming in Python. You will learn the following:

- How to create a class
- Instantiating objects
- Adding attributes to a class
- Defining methods within a class
- Passing arguments to methods
- How OOP can be used in Python for finance

OOP: Introduction

Object-oriented programming has some advantages over other design patterns. Development is faster and cheaper, with better software maintainability. This, in turn, leads to higher-quality software, which is also extensible with new methods and attributes. The learning curve is, however, steeper. The concept may be too complex for beginners. Computationally, OOP software is slower, and uses more memory since more lines of code have to be written.

Object-oriented programming is based on the imperative programming paradigm, which uses statements to change a program's state. It focuses on describing how a program should operate. Examples of imperative programming languages are C, C++, Java, Go, Ruby and Python. This stands in contrast to declarative programming, which focuses on what the computer program should accomplish, without specifying how. Examples are database query languages like SQL and XQuery, where one only tells the computer what data to query from where, but now how to do it.

OOP uses the concept of objects and classes. A class can be thought of as a 'blueprint' for objects. These can have their own attributes (characteristics they possess), and methods (actions they perform).

OOP Example

An example of a class is the class `Dog`. Don't think of it as a specific dog, or your own dog. We're describing what a dog *is* and can *do*, in general. Dogs usually have a `name` and `age`; these are instance attributes. Dogs can also `bark`; this is a method.

When you talk about a specific dog, you would have an object in programming: an object is an instantiation of a class. This is the basic principle on which object-oriented programming is based. So my dog Ozzy, for example, belongs to the class `Dog`. His attributes are `name = 'Ozzy'` and `age = '2'`. A different dog will have different attributes.

OOP in Python

Python is a great programming language that supports OOP. You will use it to define a class with attributes and methods,

offers a number of benefits compared to other programming languages like Java, C++ or R. It's a dynamic language, with high-level data types. This means that development happens much faster than with Java or C++. It does not require the programmer to declare types of variables and arguments. This also makes Python easier to understand and learn for beginners, its code being more readable and intuitive.

If you're new to Python, be sure

to take a look at DataCamp's Intro to Python for Data Science course.

How to create a class

To define a class in Python, you can use the `class` keyword, followed by the class name and a colon. Inside the class, an `__init__` method has to be defined with `def`. This is the initializer that you can later use to instantiate objects. It's similar to a constructor in Java. `__init__` must always be present! It takes one argument: `self`, which refers to the object itself. Inside the method, the `pass` keyword is used as of now, because Python expects you to type something there. Remember to use correct indentation!

```
class Dog:

    def __init__(self):
        pass
```

Note: `self` in Python is equivalent to `this` in C++ or Java.

In this case, you have a (mostly empty) `Dog` class, but no object yet. Let's create one!

Instantiating objects

To instantiate an object, type the class name, followed by two brackets. You can assign this to a variable to keep track of the object.

```
ozzy = Dog()
```

And print it:

```
print(ozzy)
```

```
<__main__.Dog object at 0x11
```

Adding attributes to a class

After printing `ozzy`, it is clear that this object is a dog. But you haven't added any attributes yet. Let's give the `Dog` class a name and age, by rewriting it:

```
class Dog:

    def __init__(self, name,
        self.name = name
        self.age = age
```

You can see that the function now takes two arguments after `self`: `name` and `age`. These then get assigned to `self.name` and `self.age` respectively. You can now now create a new `ozzy` object, with a name and age:

```
ozzy = Dog("Ozzy", 2)
```

To access an object's attributes in Python, you can use the dot notation. This is done by typing the name of the object, followed by a dot and the attribute's name.

```
print(ozzy.name)

print(ozzy.age)
```

```
Ozzy
2
```

This can also be combined in a more elaborate sentence:

```
print(ozzy.name + " is " + s
```

```
Ozzy is 2 year(s) old.
```

The `str()` function is used here to convert the `age` attribute, which is an integer, to a string, so you can use it in the `print()` function.

Define methods in a class

Now that you have a `Dog` class, it does have a name and age which you can keep track of, but it doesn't actually do anything. This is where instance methods come in. You can rewrite the class to now include a `bark()` method. Notice how the `def` keyword is used again, as well as the `self` argument.

```
class Dog:

    def __init__(self, name,
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")
```

The `bark` method can now be called using the dot notation, after instantiating a new `ozzy` object. The method should print "bark bark!" to the screen. Notice the parentheses (curly brackets) in `.bark()`. These are always used when calling a method. They're empty in this case, since the `bark()` method does not take any arguments.

```
ozzy = Dog("Ozzy", 2)
```

```
ozzy.bark()
```

```
bark bark!
```

Recall how you printed `ozzy` earlier? The code below now implements this functionality in the `Dog` class, with the `doginfo()` method. You then instantiate some objects with different properties, and call the method on them.

```
class Dog:

    def __init__(self, name,
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")

    def doginfo(self):
        print(self.name + " 
ozzy = Dog("Ozzy", 2)
skippy = Dog("Skippy", 12)
filou = Dog("Filou", 8)


ozzy.doginfo()
skippy.doginfo()
filou.doginfo()
```

```
Ozzy is 2 year(s) old.
Skippy is 12 year(s) old.
Filou is 8 year(s) old.
```

As you can see, you can call the `doginfo()` method on objects with the dot notation. The response now depends on which `Dog` object you are calling the method on.

Since dogs get older, it would be nice if you could adjust their age accordingly. Ozzy just turned 3, so let's change his age.

```
ozzy.age = 3
```

```
print(ozzy.age)
```

```
3
```

It's as easy as assigning a new value to the attribute. You could also implement this as a `birthday()` method in the `Dog` class:

```
class Dog:

    def __init__(self, name,
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")
    def doginfo(self):
        print(self.name + "

    def birthday(self):
        self.age +=1
```

```
ozzy = Dog("Ozzy", 2)
```

```
print(ozzy.age)
```

```
2
```

```
ozzy.birthday()
```

```
print(ozzy.age)
```

```
3
```

Now, you don't need to manually change the dog's age.

whenever it is its birthday, you can just call the `birthday()` method.

Passing arguments to methods

You would like for our dogs to have a buddy. This should be optional, since not all dogs are as sociable. Take a look at the `setBuddy()` method below. It takes `self`, as per usual, and `buddy` as arguments. In this case, `buddy` will be another `Dog` object. Set the `self.buddy` attribute to `buddy`, and the `buddy.buddy` attribute to `self`. This means that the relationship is reciprocal; you are your buddy's buddy. In this case, Filou will be Ozzy's buddy, which means that Ozzy automatically becomes Filou's buddy. You could also set these attributes manually, instead of defining a method, but that would require more work (writing 2 lines of code instead of 1) every time you want to set a buddy. Notice that in Python, you don't need to specify of what type the argument is. If this were Java, it would be required.

```
class Dog:

    def __init__(self, name,
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")

    def doginfo(self):
        print(self.name + " 

    def birthday(self):
        self.age +=1

    def setBuddy(self, buddy
        self.buddy = buddy
        buddy.buddy = self
```

You can now call the method with the dot notation, and pass it another `Dog` object. In this case, Ozzy's buddy will be Filou:

```
ozzy = Dog("Ozzy", 2)
filou = Dog("Filou", 8)

ozzy.setBuddy(filou)
```

If you now want to get some information about Ozzy's buddy, you can use the dot notation twice:. First, to refer to Ozzy's buddy, and a second time to refer to its attribute.

```
print(ozzy.buddy.name)
print(ozzy.buddy.age)
```

```
Filou
8
```

Notice how this can also be done for Filou.

```
print(filou.buddy.name)
print(filou.buddy.age)
```

```
Ozzy
2
```

The buddy's methods can also be called. The `self` argument that gets passed to `doginfo()` is now `ozzy.buddy`, which is `filou`.

```
ozzy.buddy.doginfo()
```

```
Filou is 8 year(s) old.
```

Example: OOP in Python for finance

An example for where Object-Oriented programming in Python might come in handy, is our Python For Finance: Algorithmic Trading tutorial. In it, Karlijn explains how to set up a trading strategy for a stock portfolio. The trading strategy is based on the moving average of a stock price. If `signals['short_mavg'][short_window:] > signals['long_mavg'][short_window:]` is fulfilled, a signal is created. This signal is a prediction for the stock's future price change. In the code below, you'll see that there is first a initialisation, followed by the moving average calculation and signal generation. Since this is not object-oriented code, it's just one big chunk that gets executed at once. Notice that we're using `aapl` in the example, which is Apple's stock ticker. If you wanted to do this for a different stock, you would have to rewrite the code.

```
# Initialize
short_window = 40
long_window = 100
signals = pd.DataFrame(index
signals['signal'] = 0.0

# Create short simple moving
signals['short_mavg'] = aapl

# Create long simple moving
signals['long_mavg'] = aapl[

# Create signals
signals['signal'][short_wind
```

```
# Generate trading orders
signals['positions'] = signa

# Print `signals`
print(signals)
```

In an object-oriented approach, you only need to write the initialisation and signal generation code once. You can then create a new object for each stock you want to calculate a strategy on, and call the `generate_signals()` method on it. Notice that the OOP code is very similar to the code above, with the addition of `self`.

```
class MovingAverage():

    def __init__(self, symbo
        self.symbol = symbol
```

```
            self.bars = bars
            self.short_window = 
            self.long_window = l

        def generate_signals(sel
            signals = pd.DataFra
            signals['signal'] = 

            signals['short_mavg'
            signals['long_mavg']

            signals['signal'][se

            signals['positions']

            return signals
```

You can now simply instantiate an object, with the parameters you want, and generate signals for it.

```
apple = MovingAverage('aapl'
print(apple.generate_signals
```

Doing this for another stock becomes very easy. It's just a matter of instantiating a new object with a different stock symbol.

```
microsoft = MovingAverage('m
print(microsoft.generate_sig
```

Congratulations!

You now know how to declare classes and methods, instantiate objects, set their attributes and call instance methods. These skills will come in handy during your future career as a data scientist. If you want to expand the key concepts that you need to further work with Python, be sure to check out our Intermediate Python for Data Science course.

With OOP, your code will grow in complexity as your program gets larger. You will have different classes, subclasses, objects, inheritance, instance methods, and more. You'll want to keep your code properly structured and readable. To do so, it is advised to follow design patterns. These are design principles that represent a set of guidelines to avoid bad design. They each represent a specific problem that often reoccurs in OOP, and describe the solution to that problem, which can then be used repeatedly. These OOP design patterns can be classified in several categories: creational patterns, structural patterns and behavioral patterns. An example of a creational pattern is the singleton, which should be used when you want to make sure that only one instance of a class can be created. An iterator, which is used to loop over all objects in a collection, is an example of a behavioral pattern. A great resource for design patterns is oodesign.com. If you're more into books, I would recommend you to read Design Patterns: Elements of Reusable Object-Oriented Software.

COMMENTS

Dea Venditama
thanks for this article.. its very helpfull for me to understand the OOP concept in python

▲ 7 REPLY | 28/03/2018 12:59 PM

Théo Vanderheyden

I'm glad I could help :)

▲ REPLY | 28/03/2018 02:50 PM

Anas Helios

this article is so good, it would be great if we found this courses on datacamp library courses

▲ 14 REPLY | 29/03/2018 04:48 AM

Ketan Patel

Thanks a lot. I have been looking for such article on python OOP. wish more on the same topic.

thanks

▲ 5 REPLY | 29/03/2018 06:01 AM

Gabriel Koch

It helped me, but I couldn`t quite understand the **Passing arguments to methods** as much as I would like.

Can anyone help me understand the buddy.buddy?

▲ 2 REPLY | 29/03/2018 06:17 PM

simonriezebos

That might be unclear because the first buddy refers to something different than the second buddy, does this help?

```
def setBuddy(self, buc
        self.buddy = k
        buddy_of_self.
```

When you use the setBuddy Method you have to pass a buddy Dog as an argument.

▲ : REPLY | 03/04/2018 03:02 PM

Abhi Rajan

Thanks for the article. I typically write functions and leave things at that. This article was really helpful. I had one question, in the MovingAverage class, when initializing stuff, why choose to init both symbol and bars (also why the variable name "bars"?). As far as I can make out, only bars is necessary to run the method.

Also I think you want to pass bars to the generate_signals method - def generate_signals(self, bars)

▲ 1 REPLY | 03/04/2018 12:24 PM

Boggavarapu Rss Srinivas Gupta

Thanks for author for writing article in simple way, which can easily understood. My opinion is this article is mainly focused on basics of classes but some advanced concepts like inheritance which has lion's share in oops concept was not discussed.

▲ 2 REPLY | 03/04/2018 01:55 PM

Ajay Manwani

Thanks ! Very Nice Article !!

▲ 3 REPLY | 15/04/2018 08:47 AM

Yogesh Shetty

▲ 1 REPLY | 22/04/2018 10:18 AM