Previous: Python in VSCode

Next: Python Constructor ›

Home » Classes and Objects in Python

Classes and Objects in Python

May 17, 2022

The Python class and Python objects are a crucial part of the language. You can't properly learn Python without understanding Python classes and objects. In this chapter, you will learn:

- How in Python everything is an object
- To define your own Python class
- Create objects based on a class
- What inheritance is

When you're just creating small scripts, chances are you don't need to create your own Python classes. But once you start creating larger applications, objects and classes allow you to organize your code naturally. A good understanding of objects and classes will help you understand the language itself much better.

Table of Contents [hide]

Beginners Python Course (2023) **Modules, Packages, And Virtual Environments (2023)**

Python objects: a look under the hood

Before we dive into all the details, let's start by taking a look under the hood. I do this because I believe it will give you a much better understanding of these concepts. Don't let the length of this page discourage you. After reading it thoroughly, and trying the examples yourself, you should have a good understanding of classes and objects in Python.

OK; let's dive in! You probably know the built-in `len()` function. It simply returns the length of the object you give it. But what is the length of, say, the number five? Let's ask Python:

```
>>> len(5)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

I love errors because they illustrate how Python works internally. In this case, Python is telling us that 5 is an object, and it has no `len()`. In Python, everything is an object. Strings, booleans, numbers, and even Python functions are objects. We can inspect an object in the REPL using the built-in function `dir()`. When we try `dir` on the number five, it reveals a big list of functions that are part of any object of type number:

```
>>> dir(5)
['__abs__', '__add__',
'__and__', '__bool__',
'__ceil__', '__class__',
...
'__str__', '__sub__',
 '__subclasshook__', '__truediv__',
 '__trunc__', '__xor__',
'bit_length', 'conjugate',
'denominator', 'from_bytes',
 'imag', 'numerator',
'real', 'to_bytes']
```

I truncated the list a little for the sake of clarity.

The list starts with these weirdly named functions containing underscores, like `__add__`. These are called magic methods, or dunder (short for double underscore) methods. If you look closely, you'll see that there's no `__len__` dunder method for objects of type `int`. That's how Python's `len()` function knows that a number does not have a length. All `len()` does, is call the `__len__()` method on the object you offered it. That's also why Python complained that "objects of type 'int' have no len()."

What are Python Methods?

I casually introduced the word methods here. Let me define it more formally:

Method
When a function is part of an object or Python class, we call it a method.

Strings do have a length, so a string must have a `len` method, right? Let's find out!

```
>>> dir("test")
```

```
['__add__', '__class__',
'__contains__', '__delattr__',
'__dir__', '__doc__',
'__eq__', '__format__',
'__ge__', '__getattribute__',
'__getitem__', '__getnewargs__',
'__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__',
'__new__', '__reduce__',
'__reduce_ex__', '__repr__',
'__rmod__', '__rmul__',
'__setattr__', '__sizeof__',
'__str__', '__subclasshook__',
'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith',
'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate',
'upper', 'zfill']
```

Yup, there it is. And since this is a method, we can call it too:

```
>>> "test".__len__()
4
```

This is equivalent to `len("test")` but a lot less elegant, so don't do this. It's just to illustrate how this stuff works.

There's also a list of other, less magical methods that `dir()` revealed to us. Feel free to try a few, like `islower`:

```
>>> "test".islower()
True
```

This method checks if the entire string is lower-case, which it is, so Python returns the boolean `True`. Some of these methods require one or more arguments, like replace:

```
>>> 'abcd'.replace('a', 'b')
'bbcd'
```

It replaces all occurrences of 'a' with 'b'.

Sometimes programming might feel a bit like doing magic, especially when you're just starting out. But once you take a peek under the hood and see how things actually work, a lot of that magic is gone. Let's continue and find out what objects really are, and how they are defined.

What is a Python object?

Now that we've used objects and know that everything in Python is an object, it's time to define what an object is:

Object
An object is a collection of data (variables) and methods that operate on that data. Objects are defined by a Python class.

Objects and object-oriented programming are concepts that became popular in the early 1990s. Early computer languages, like C, did not have the concept of an object. However, it turned out that objects are an easy-to-understand paradigm for humans. Objects can be used to model many real-life concepts and situations. Most, if not all, new languages have the concept of objects these days. So what you're about to learn will conceptually apply to other languages too: this is fundamental computer science.

What is a Python class?

Since objects are the building blocks of the Python language, it's only logical that you can create objects yourself too. If you want to create your own type of object, you first need to define its methods and what data it can hold. This blueprint is called a class.

Class
A class is the blueprint for one or more objects

All Python objects are based on a class. When we create an object, we call this 'creating an instance of a class'. Strings, numbers, and even booleans are instances of a class too. Let's explore with the built-in function `type`:

```
>>> type('a')
<class 'str'>
>>> type(1)
<class 'int'>
type(True)
<class 'bool'>
```

Apparently, there are classes called `str`, `int`, and `bool`. These are some of Python's native classes, but, as I said, we can build our own types of classes too!

Creating a Python class

No tutorial is complete without a car analogy, so let's make a Python class that represents a car. It's a lot to type in, and you have to start over on each mistake. Feel free to try, but I understand if you want to take a shortcut. Just copy and paste the following class into your Python REPL. Make sure to **hit enter twice after pasting** it:

```
class Car:
    speed = 0
    started = False
    def start(self):
        self.started = True
        print("Car started, let's ride!")
    def increase_speed(self, delta):
        if self.started:
            self.speed = self.speed + delta
            print('Vroooooom!')
        else:
            print("You need to start the car first")
    def stop(self):
        self.speed = 0
        print('Halting')
```

Create a Python object

Don't worry, we'll go over the class definition step by step, but let's first create and use a Python object of type Car:

```
>>> car = Car()
>>> car.increase_speed(10)
You need to start the car first
>>> car.start()
Car started, let's ride!
>>> car.increase_speed(40)
Vroooooom!
```

If you prefer, you can also use the following crumb to play around with our newly created Car class:

An object in Python is always an instance of a class. One class can have many instances. We just created an instance of class `Car`, with `Car()`, and assigned it to the (lowercase) variable `car`. Creating an instance looks like calling a function; you'll learn why later on.

Next, we call one of our car object methods: trying to increase its speed while it's not started yet. Oops! Only after starting the car, we can increase its speed and enjoy the noise it makes.

Now let's go over our Car class step by step:

- A class in Python is defined using the class statement, followed by the class's name (Car). We start an indented block of code with the colon.
- We defined two variables, speed and started. This is data that all instances of this class will have.
- Next, we defined three methods that operate on the variables.

In the definitions of these methods, we encounter something peculiar: they all have an argument called self as their first argument.

What Is self in Python?

Honestly, this is one of Python's less elegant language constructs if you ask me.

Remember when we were calling the methods on our car object, like `car.start()`? We didn't have to pass the `self` variable, even though `start()` is defined as `start(self)` inside the class.

This is what's happening:

- When we call a method on a Python object, Python automatically fills in the first variable, which we call self by convention.
- This first variable is a reference to the object itself, hence its name.
- We can use this variable to reference other instance variables and functions of this object, like `self.speed` and `self.start()`.

So only inside the Python class definition, we use `self` to reference variables that are part of the instance. To modify the `started` variable that's part of our class, we use `self.started` and not just `started`. By using `self`, it's made abundantly clear that we are operating on a variable that's part of this instance and not some other variable that is defined outside of the object and happens to have the same name.

Creating Multiple Python Objects

Since a Python class is just a blueprint, you can use it to create multiple objects, just like you can build multiple identical-looking cars. They all behave similarly, but they all have their own data that is not shared between objects:

```
>>> car1 = Car()
>>> car2 = Car()
>>> id(car1)
139771129539104
>>> id(car2)
139771129539160
```

We created two car objects here, car1 and car2, and used the built-in method id() to get their ids. Each object in Python has a unique identifier, so we just proved that we created two different objects from the same class. We can use them independently:

```
>>> car1.start()
Car started, let's ride!
>>> car1.increase_speed(10)
'Vrooom!'
>>> car1.speed
10
>>> car2.speed
0
```

We just started `car1` and increased its speed, while `car2` is still halted. Inspection of the speeds confirms these are different cars with different states!

Keep learning

- You might be interested in Python data classes too
- How to return multiple values from a function
- The official documention on classes

‹ Previous: Python in VSCode

Next: Python Constructor