Python’s Functions Are First-Class

by Dan Bader — Get free updates of new posts here.

Python’s functions are first-class objects. You can assign them to variables, store them in data structures, pass them as arguments to other functions, and even return them as values from other functions.

Grokking these concepts intuitively will make understanding advanced features in Python like lambdas and decorators much easier. It also puts you on a path towards functional programming techniques.

In this tutorial I’ll guide you through a number of examples to help you develop this intuitive understanding. The examples will build on top of one another, so you might want to read them in sequence and even to try out some of them in a Python interpreter session as you go along.

Wrapping your head around the concepts we’ll be discussing here might take a little longer than expected. Don’t worry—that’s completely normal. I’ve been there. You might feel like you’re banging your head against the wall, and then suddenly things will “click” and fall into place when you’re ready.

Throughout this tutorial I’ll be using this `yell` function for demonstration purposes. It’s a simple toy example with easily recognizable output:

```
def yell(text):

    return text.upper() + '!'
```

```
>>> yell('hello')

'HELLO!'
```

Functions Are Objects

All data in a Python program is represented by objects or relations between objects. Things like strings, lists, modules, and functions are all objects. There's nothing particularly special about functions in Python.

Because the `yell` function is an *object* in Python you can assign it to another variable, just like any other object:

```
>>> bark = yell
```

This line doesn't call the function. It takes the function object referenced by `yell`and creates a second name pointing to it, `bark`. You could now also execute the same underlying function object by calling `bark`:

```
>>> bark('woof')

'WOOF!'
```

Function objects and their names are two separate concerns. Here's more proof: You can delete the function's original name (`yell`). Because another name (`bark`) still points to the underlying function you can still call the function through it:

```
>>> del yell



>>> yell('hello?')

NameError: "name 'yell' is not defined"



>>> bark('hey')

'HEY!'
```

By the way, Python attaches a string identifier to every function at creation time for debugging purposes. You can access this internal identifier with the `__name__`attribute:

```
>>> bark.__name__

'yell'
```

While the function's `__name__` is still "yell" that won't affect how you can access it from your code. This identifier is merely a debugging aid. A *variable pointing to a function* and the *function itself* are two separate concerns.
(Since Python 3.3 there's also `__qualname__` which serves a similar purpose and provides a *qualified name* string to disambiguate function and class names.)

Functions Can Be Stored In Data Structures

As functions are first-class citizens you can store them in data structures, just like you can with other objects. For example, you can add functions to a list:

```
>>> funcs = [bark, str.lower, str.capitalize]

>>> funcs

[<function yell at 0x10ff96510>,

 <method 'lower' of 'str' objects>,

 <method 'capitalize' of 'str' objects>]
```

Accessing the function objects stored inside the list works like it would with any other type of object:

```
>>> for f in funcs:

...     print(f, f('hey there'))

<function yell at 0x10ff96510> 'HEY THERE!'

<method 'lower' of 'str' objects> 'hey there'

<method 'capitalize' of 'str' objects> 'Hey there'
```

You can even call a function object stored in the list without assigning it to a variable first. You can do the lookup and then immediately call the resulting "disembodied" function object within a single expression:

```
>>> funcs[0]('heyho')

'HEYHO!'
```

Functions Can Be Passed To Other Functions

Because functions are objects you can pass them as arguments to other functions. Here's a `greet` function that formats a greeting string using the function object passed to it and then prints it:

```
def greet(func):

    greeting = func('Hi, I am a Python program')

    print(greeting)
```

You can influence the resulting greeting by passing in different functions. Here's what happens if you pass the `yell` function to `greet`:

```
>>> greet(yell)

'HI, I AM A PYTHON PROGRAM!'
```

Of course you could also define a new function to generate a different flavor of greeting. For example, the following `whisper` function might work better if you don't want your Python programs to sound like Optimus Prime:

```
def whisper(text):

    return text.lower() + '...'



>>> greet(whisper)

'hi, i am a python program...'
```

The ability to pass function objects as arguments to other functions is powerful. It allows you to abstract away and pass around *behavior* in your programs. In this example, the `greet` function stays the same but you can influence its output by passing in different *greeting behaviors*. Functions that can accept other functions as arguments are also called *higher-order functions*. They are a necessity for the functional programming style.

The classical example for higher-order functions in Python is the built-in `map`function. It takes a function and an iterable and calls the function on each element in the iterable, yielding the results as it goes along.
Here's how you might format a sequence of greetings all at once by *mapping* the `yell` function to them:

```
>>> list(map(yell, ['hello', 'hey', 'hi']))

['HELLO!', 'HEY!', 'HI!']
```

`map` has gone through the entire list and applied the `yell` function to each element.

Functions Can Be Nested

Python allows functions to be defined inside other functions. These are often called *nested functions* or *inner functions*. Here's an example:

```
def speak(text):

    def whisper(t):

        return t.lower() + '...'

    return whisper(text)



>>> speak('Hello, World')

'hello, world...'
```

Now, what's going on here? Every time you call `speak` it defines a new inner function `whisper` and then calls it.
And here's the kicker—`whisper` *does not exist* outside `speak`:

```
>>> whisper('Yo')

NameError: "name 'whisper' is not defined"



>>> speak.whisper

AttributeError: "'function' object has no attribute 'whisper'"
```

But what if you really wanted to access that nested `whisper` function from outside `speak`? Well, functions are objects—you can *return* the inner function to the caller of the parent function.
For example, here's a function defining two inner functions. Depending on the argument passed to top-level function it selects and returns one of the inner functions to the caller:

```
def get_speak_func(volume):

    def whisper(text):

        return text.lower() + '...'

    def yell(text):

        return text.upper() + '!'

    if volume > 0.5:

        return yell
    else:

        return whisper
```

Notice how `get_speak_func` doesn't actually *call* one of its inner functions—it simply selects the appropriate function based on the `volume` argument and then returns the function object:

```
>>> get_speak_func(0.3)

<function get_speak_func.<locals>.whisper at 0x10ae18>



>>> get_speak_func(0.7)

<function get_speak_func.<locals>.yell at 0x1008c8>
```

Of course you could then go on and call the returned function, either directly or by assigning it to a variable name first:

```
>>> speak_func = get_speak_func(0.7)

>>> speak_func('Hello')

'HELLO!'
```

Let that sink in for a second here… This means not only can functions *accept behaviors* through arguments but they can also *return behaviors*. How cool is that?

You know what, this is starting to get a little loopy here. I'm going to take a quick coffee break before I continue writing (and I suggest you do the same.)

Functions Can Capture Local State

You just saw how functions can contain inner functions and that it's even possible to return these (otherwise hidden) inner functions from the parent function.

Best put on your seat belts on now because it's going to get a little crazier still—we're about to enter even deeper functional programming territory. (You had that coffee break, right?)

Not only can functions return other functions, these inner functions can also *capture and carry some of the parent function's state* with them.

I'm going to slightly rewrite the previous `get_speak_func` example to illustrate this. The new version takes a "volume" *and* a "text" argument right away to make the returned function immediately callable:

```
def get_speak_func(text, volume):
    def whisper():

        return text.lower() + '...'

    def yell():

        return text.upper() + '!'

    if volume > 0.5:

        return yell

    else:

        return whisper



>>> get_speak_func('Hello, World', 0.7)()

'HELLO, WORLD!'
```

Take a good look at the inner functions `whisper` and `yell` now. Notice how they no longer have a `text` parameter? But somehow they can still access the `text`parameter defined in the parent function. In fact, they seem to *capture* and “remember” the value of that argument. Functions that do this are called *lexical closures* (or just *closures*, for short). A closure remembers the values from its enclosing lexical scope even when the program flow is no longer in that scope.

In practical terms this means not only can functions *return behaviors* but they can also *pre-configure those behaviors*. Here’s another bare-bones example to illustrate this idea:

```
def make_adder(n):

    def add(x):

        return x + n

    return add



>>> plus_3 = make_adder(3)

>>> plus_5 = make_adder(5)
>>> plus_3(4)

7

>>> plus_5(4)

9
```

In this example `make_adder` serves as a *factory* to create and configure “adder” functions. Notice how the “adder” functions can still access the `n` argument of the `make_adder` function (the enclosing scope).

Objects Can Behave Like Functions

Object’s aren’t functions in Python. But they can be made *callable*, which allows you to *treat them like functions* in many cases.

If an object is callable it means you can use round parentheses `()` on it and pass function call arguments to it. Here’s an example of a callable object:

```
class Adder:

    def __init__(self, n):

         self.n = n

    def __call__(self, x):

        return self.n + x



>>> plus_3 = Adder(3)

>>> plus_3(4)

7
```

Behind the scenes, “calling” an object instance as a function attempts to execute the object’s `__call__` method.
Of course not all objects will be callable. That’s why there’s a built-in `callable`function to check whether an object appears callable or not:

```
>>> callable(plus_3)

True

>>> callable(yell)
```

```
True

>>> callable(False)

False
```

Key Takeaways

- Everything in Python is an object, including functions. You can assign them to variables, store them in data structures, and pass or return them to and from other functions (first-class functions.)
- First-class functions allow you to abstract away and pass around behavior in your programs.
- Functions can be nested and they can capture and carry some of the parent function’s state with them. Functions that do this are called *closures*.
- Objects can be made callable which allows you to treat them like functions in many cases.