Python's Functions Are First-Class

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.

```
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!'
```

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
```

Python Decorators

Python Decorator Basics

Now, what are decorators really? They “decorate” or “wrap” another function and let you execute code before and after the wrapped function runs.

Decorators allow you to define reusable building blocks that can change or extend the behavior of other functions. And they let you do that without permanently modifying the wrapped function itself. The function’s behavior changes only when it’s *decorated*.

Now what does the implementation of a simple decorator look like? In basic terms, a decorator is *a callable that takes a callable as input and returns another callable*.

The following function has that property and could be considered the simplest decorator one could possibly write:

```
def null_decorator(func):

    return func
```

As you can see, `null_decorator` is a callable (it’s a function), it takes another callable as its input, and it returns the same input callable without modifying it. Let’s use it to *decorate* (or *wrap*) another function:

```
def greet():

    return 'Hello!'



greet = null_decorator(greet)



>>> greet()
```

```
'Hello!'
```

In this example I've defined a `greet` function and then immediately decorated it by running it through the `null_decorator` function. I know this doesn't look very useful yet (I mean we specifically designed the null decorator to be useless, right?) but in a moment it'll clarify how Python's decorator syntax works.
Instead of explicitly calling `null_decorator` on `greet` and then reassigning the `greet` variable, you can use Python's `@` syntax for decorating a function in one step:

```
@null_decorator

def greet():

    return 'Hello!'



>>> greet()

'Hello!'
```

Putting an `@null_decorator` line in front of the function definition is the same as defining the function first and then running through the decorator. Using the `@`syntax is just *syntactic sugar*, and a shortcut for this commonly used pattern.

Applying Multiple Decorators to a Single Function

Perhaps not surprisingly, you can apply more than one decorator to a function. This accumulates their effects and it's what makes decorators so helpful as reusable building blocks.

Here's an example. The following two decorators wrap the output string of the decorated function in HTML tags. By looking at how the tags are nested you can see which order Python uses to apply multiple decorators:

```
def strong(func):

    def wrapper():
```

```
        return '<strong>' + func() + '</strong>'

    return wrapper



def emphasis(func):

    def wrapper():

        return '<em>' + func() + '</em>'

    return wrapper
```

Now let's take these two decorators and apply them to our `greet` function at the same time. You can use the regular `@` syntax for that and just "stack" multiple decorators on top of a single function:

```
@strong

@emphasis

def greet():

    return 'Hello!'
```

What output do you expect to see if you run the decorated function? Will the `@emphasis` decorator add its `<em>` tag first or does `@strong` have precedence? Here's what happens when you call the decorated function:

```
>>> greet()

'<strong><em>Hello!</em></strong>'
```

This clearly shows in what order the decorators were applied: from *bottom to top*.

*args and **kwargs in Python

`*args` to send a variable-length argument list to our function, we were able to pass in as many arguments as we wished into the function calls.

With `*args` you can create more flexible code that accepts a varied amount of non-keyworded arguments within your function.

```
# *args example
def multiply(*args):
    z = 1
    for num in args:
        z *= num
    print(z)

multiply(4, 5)
multiply(10, 9)
multiply(2, 3, 4)
multiply(3, 5, 10, 6)

#**kwargs example

def print_kwargs(**kwargs):
        print(kwargs)

print_kwargs(kwargs_1="Shark", kwargs_2=4.5, kwargs_3=True)


Output:
{'kwargs_3': True, 'kwargs_2': 4.5, 'kwargs_1': 'Shark'}
```

Depending on the version of Python 3 you are currently using, the dictionary data type may be unordered. In Python 3.6 and above, you'll receive the key-value pairs in order, but in earlier versions, the pairs will be output in a random order.