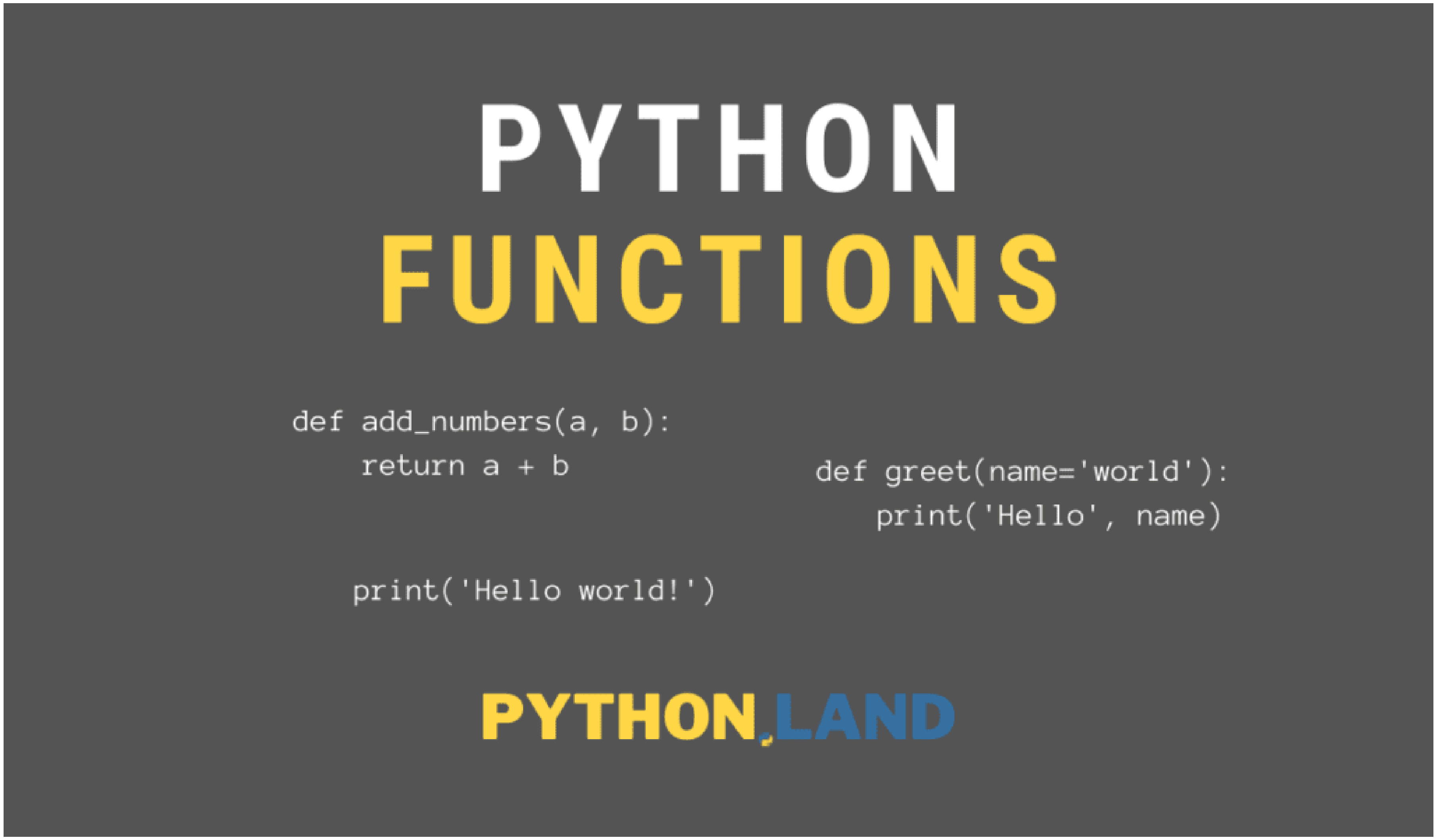

```
def add_numbers(a, b):
    return a + b

def greet(name='world'):
    print('Hello', name)

print('Hello world!')
```

Python Function: The Basics Of Code Reuse

August 8, 2022

The function is a crucial concept in the world of programming. In this article, we'll explore Python functions. You'll learn why they are so important, how to define functions with Python's def keyword, how to call functions, and we'll learn about a topic that arises when using functions: variable scope.

Table of Contents [hide]

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

What is a function in Python?

Let's define what a function is, exactly:

Function
A Python function is a named section of a program that performs a specific task and, optionally, returns a value

Functions are the real building blocks of any programming language. We define a Python function with the `def` keyword. But before we start doing so, let's first go over the advantages of functions, and let's look at some built-in functions that you might already know.

Advantages of using functions

Code reuse

A Python function can be defined once and used many times. So it aids in code reuse: you don't want to write the same code more than once.

Functions are a great way to keep your code short, concise, and readable. By giving a function a well-chosen name, your code will become even more readable because the function name directly explains what will happen. This way, others (or future you) can read your code, and without looking at all of it, understand what it's doing anyway because of the well-chosen function names.

Other forms of code reuse are explained later on. E.g., you can also group code into modules and packages.

Parameters

Functions accept a parameter, and you'll see how this works in a moment. The big advantage here, is that you can alter the function's behavior by changing the parameter.

Return values

A function can return a value. This value is often the result of some calculation or operation. In fact, a Python function can even return multiple values.

Built-in Python functions

Before we start defining functions ourselves, we'll look at some of Python's built-in functions. Let's start with the most well-known built-in function, called `print`:

```
>>> print('Hello, readers!')
Hello, readers!
>>> print(15)
15
```

Print takes an argument and prints it to the screen.

As stated in our definition, functions can optionally return a value. However, `print` does not return anything. Because it prints something to the screen, it might look like it does, but it doesn't. We can check this by assigning the result of a print statement to a Python variable:

```
>>> result = print('Hello')
Hello
>>> print(result)
None
```

None is a special type of value in Python, basically meaning 'nothing.'

Another built-in function, that does return a value, is `len()`. It returns the length of whatever you feed it:

```
>>> mylength = len('Hello')
>>> print(mylength)
5
```

Creating a Python function

Now that we know how to use a function let's create a simple one ourselves. To do so, we use Python's `def` keyword:

```
>>> def say_hi():
...     print('Hi!')
...
>>> say_hi()
Hi!
```

It's just a few lines, but a lot is going on. Let's dissect this:

- First of all, we see the keyword `def`, which is Python's keyword to define a function.
- Next comes our function name, say_hi.
- Then we encounter two parentheses, (), which indicate that this function does not accept any parameters (unlike `print` and `len`).
- We end the line with a colon (:)
- And finally, we bump into a feature that sets Python apart from many other programming languages: indentation.

Indentation

Python uses indentation to distinguish blocks of code that belong together. Consecutive lines with equal indentation are part of the same block of code.

To tell Python that the following lines are the body of our function, we need to indent them. You indent lines with the **TAB key** on your keyboard. In Python, it's good style to use four spaces for indentation. The entire Python community does so. If you hit the TAB key, the Python interactive shell (REPL) and all decent editors will automatically indent with four spaces.

Back to our function: it has only one line of code in its body: the print command. After it, we hit enter one extra time to let the Python REPL know that this is the end of the function. This is important. A function must always be followed by an empty line to signal the end of the function. Finally, we can call our function with `say_hi()`.

Python function parameters and arguments

We can make this more interesting by allowing an argument to be passed to our function. Again we define a function with def, but we add a variable name between the parentheses:

```
>>> def say_hi(name):
...     print('Hi', name)
...
>>> say_hi('Erik')
Hi Erik
```

Our function now accepts a value, which gets assigned to the variable name. We call such variables the **parameter**, while the actual value we provide ('Erik') is called the **argument**.

Parameters and arguments
A Python function can have parameters. The values we pass through these parameters are called arguments.

As you can see, `print()` accepts multiple arguments, separated by a comma. This allows us to print both 'hi' and the provided name. For our convenience, `print()` automatically puts a space between the two strings.

Python function with multiple arguments

Let's make this even wilder and define a function with multiple arguments:

```
>>> def welcome(name, location):
...     print("Hi", name, "welcome to", location)
...
>>> welcome('Erik', 'this tutorial')
Hi Erik welcome to this tutorial
```

It's not that hard; you just add more arguments to the function definition, separated by commas.

Returning from a function

A function runs until the end of that function, after which Python returns to where the function was called from. In the following example, I want you to predict the output before you run it:

Predict the output of this program

Did your expectations match with reality? If so, you have a good understanding of how a function call works. Some people at this point expect to see the text “Let’s greet the entire world” twice. If you are one of them, don’t worry and keep reading.

Python keeps track of the point where the function is called. In our case, it’s at line 5. Once called, Python runs the block of code inside the function line by line until it reaches the end of that function. And once the end of the function is reached, *Python jumps back to the line right after where the function was called from*: line 6!

In short: a function call is not a jump or ‘go to’, but a call to a piece of reusable code, which returns to where it was called from. A function call is also an expression: most functions return a value.

Returning values

So far, our function only printed something and returned nothing. What makes functions a lot more usable is the ability to return a value. Let’s see an example of how a Python function can return a value:

This is how you return a value from a Python function

As you can see, we use the keyword `return` to return a value from our function. Functions that return a value can be used everywhere we can use an expression. In the above example, we can assign the returned value to the variable `result`.

We could have used the function in an if statement as well. For example:

```
if add(1, 1) == 2:
    print("That's what you'd expect!")
```

Empty return statement

If your function does not return anything, but you still want to return from the function, you can use an empty return statement. Here’s a silly example:

```
def optional_greeter(name):
    if name.startswith('X'):
        # We don't greet people with weird names :p
        return

    print('Hi there, ', name)
optional_greeter('Xander')
```

This is an interesting pattern; I call it returning early. I'd like to return early because the alternative is using blocks of if… else statements:

```
def optional_greeter(name):
    if name.startswith('X'):
        # We don't greet people with weird names :p
        pass
    else:
        print('Hi there, ', name)
optional_greeter('Xander')
```

Which one do you feel looks cleaner? I'd say the first because it requires less indented code. And although the difference is small with such a small example, this starts to add up when you have bigger chunks of code.

Variable scope

The variable name only exists inside our function. We say that the scope of the variable name is limited to the function `say_hi`, meaning it doesn't exist outside of this function.

Scope
The visibility of a variable is called scope. The scope defines which parts of your program can see and use a variable.

If we define a variable at the so-called top level of a program, it is visible in all places.

Let's demonstrate this:

```
>>> def say_hi():
...     print("Hi", name)
...     answer = "Hi"
...
>>> name = 'Erik'
>>> say_hi()
Hi Erik
```

```
>>> print(answer)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'answer' is not defined
```

`say_hi` was able to use the variable `name`, as expected, because it's a top-level variable: it is visible everywhere. However, `answer`, defined inside say_hi, is not known outside of the function and causes a `NameError`. Python gives us an informative and detailed error: "name 'answer' is not defined."

Default values and named parameters

A compelling Python feature is the ability to provide default values for the parameters:

```
>>> def welcome(name='learner', location='this tutorial'):
...     print("Hi", name, "welcome to", location)
...
>>> welcome()
Hi learner welcome to this tutorial
>>> welcome(name='John')
Hi John welcome to this tutorial
>>> welcome(location='this epic tutorial')
Hi learner welcome to this epic tutorial
>>> welcome(name='John', location='this epic tutorial')
Hi John welcome to this epic tutorial
```

Or, if you prefer, use the interactive example below:

Demonstration of default values and named parameters

Because our parameters have a default value, you don't have to fill them in. If you don't, the default is used. If you do, you override the default with your own value.

Calling Python functions while explicitly naming the parameters differs from what we did up until now. These parameters are called named parameters because we specify both the name and the value instead of just the value. Thanks to these named parameters, the order in which we supply them doesn't matter. If you think about it, it's the natural and only way to make default values useful.

If you don't want to use named parameters, you can. When you rely on position instead of names, you provide what we call positional parameters. The position matters, as can be seen below:

```
>>> def welcome(name='learner', location='this tutorial'):
...     print("Hi", name, "welcome to", location)
...
>>> welcome('Erik', 'your home')
Hi Erik welcome to your home
```

For now, you've learned enough about functions to continue with the tutorial. If you like to learn more, read my Python functions deep dive.

Get certified with our courses

Learn Python properly through small, easy-to-digest lessons, progress tracking, quizzes to test your knowledge, and practice sessions. Each course will earn you a downloadable course certificate.

‹ Previous: Python For Loop and While Loop

Next: Your First Python Program