Menu

Previous Next

Unit 1 / Lesson 3 / Assignment 1

Overview of Functions in Python

Estimated Time: **1 hour**

Although we haven't formally introduced functions yet, you've used them in several places, earlier in this unit. For instance, we've used input(), len(), .format(), and several other built-in Python functions so far.

We were able to use these functions without knowing anything about how they were implemented. All we needed to know was their *method signature*. *Method signature* refers to the arguments required by a function – in other words, the stuff we need to put between the parentheses when the function is called. And herein lies the power of functions – you can use a function without knowing anything about its implementation. You only need to know what inputs it requires, and what outputs it provides. This allows you to write what programmers call loosely coupled code.

Creating Custom Functions

At their simplest, functions provide a way of grouping together and naming a block of code that does something, supplying that code with arguments/parameters, and then doing something with those arguments/parameters.

Here's how you define a function:

```
def subtractor(a, b):
    """I subtract b from a and return the result"""
    print("I'm a function. My name is {}".format(subtractor.__name__))
    print("I'm about to subtract {} and {}\n\n".format(a,b))
    return a - b  # i output a value by using the return statement


if __name__ == '__main__':
    subtractor(3, 2)
```

You can clone this code example here or else copy and paste into a new file in your text editor. Run the function from the command line with `python3 function_demo.py` .

A few things to note about this function:

- To define a function, you use the `def` statement like in this example. `def` is followed by the name of your function, followed by a tuple of any arguments the function takes.
- This function's name is `subtractor` – it takes two parameters: a and b.
- That triple quoted text on the second line of the function is called a docstring. These optionally appear as the first line after the def statement and are used to provide a short descriptive statement of what the function does. It's considered good practice to provide a docstring for each function you write. This orients you or anyone else looking at the code, so they know what a function does without looking at its implementation. Docstrings also appear if you type help(my_function) from the Python interpreter.
- The first print call of this function prints out the name of the function. It does this by using `.format()` and substituting in the value of the `.__name__` attribute of the function. For any given function, you can access its name with this attribute.

- This function has a return statement which outputs a value.
- Check out that strange `if __name__ == '__main__':` code at the bottom of the program. `if __name__ == '__main__':` is how you define a routine to be executed when (and only when) a Python script is run from the command line. In this case, we're telling Python to call our subtractor function with the arguments 3 and 2 if it gets called from the command line. Alternatively, we can open an interactive Python console and import the function and use it there.

Now you know how to define a function. The next thing to know about functions is that just like strings, numbers, lists, and dictionaries, they can be assigned to variables. From the command line, navigate to the folder containing function_demo.py. Then start the interactive console by entering `python3` . From inside the console, enter `from function_demo import subtractor` . Now we can access our subtractor function inside our interactive console session. Next, enter `my_func = subtractor` . If you check the type of my_func, you'll see that it's a function object. You can call the function like this `my_func(3,2)` which would output 1.

Note

If you want to assign a function to a variable, you must omit the parentheses. If you include the parentheses, you'll actually call the function, instead of assigning it to a variable.

In this first example, our function has a return statement and it takes arguments, but not all functions have to do this. Consider the following function, which only prints something:

```
def print_function():
    """I'm also a function, but I don't take any parameters"""
    print("I'm {}, and I'm printing now".format(print_function.__name__
if __name__ == '__main__':
   print_function()
```

Functions Can Call other Functions

It’s also possible for a function to call another function. If print_function and subtractor have been previously defined, you could define the following function:

```
def function3(a=1, b=1):
  """I'm a function that calls other functions """
  print("I'm {} and I'm about to call subtractor function".format(func
  total = subtractor(a,b)
  print("I'm {} and I'm about to call print_function".format(function3
  print_function()
  print("I'm {} and I'm about return total".format(function3.__name__)
  return total

if __name__ == '__main__':
  total = function3()
  print("total is {}".format(total))
```

This function prints a message that it's about to call subtractor, and then it assigns a value to total by calling subtractor. It then prints a statement that it’s about to call print_function, then calls print_function. Finally, it returns total.

Note in this function, we’ve assigned *default values* for a and b with `(a=1, b=1)`. This means that if we call this function but do not supply parameters, a and b will both default to 1. This is a handy way Python lets us supply default values for function arguments.

Functions can have side-effects

So far the functions that we've looked at don't have side effects. We have passed some variables into a function, we've got a return value from the function and it hasn't changed anything else in your application. This is often a good thing as it makes it very easy to figure out how your system will work and you aren't surprised when one part of your application unexpectedly changes another part.

In Python, however, things aren't always that simple. Sometimes functions do have side-effects and you need to know how they will affect your code.

Let's create an example to demonstrate when side-effects take place:

```
def side_effect_test(value):
    # Do something to modify the value
    value[1] = "orange"
    print("Inside the function, the value becomes {}".format(value))

if __name__ == "__main__":
    # Create the value
    value = ["red", "green", "blue"]

    print("Outside the function, the value starts as {}".format(value)

    side_effect_test(value)

    print("Outside the function, the value is now {}".format(value))
```

Try running the example. Notice how changes to the value inside the function affect its value outside of the function: this is a side-effect of the function. Now try modifying your code so that `value` is one of the following types:

- A number
- A dictionary
- A string
- A boolean

- A tuple

Which types exhibit side effects and which don't? Can you identify a pattern which will allow you to work out whether a variable passed into a function can be modified by that function?

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Completed

Previous

Next