Theory: Declaring a function

17 minutes 12 / 13 problems solved

Start practicing

Often enough, built-in functions cannot suffice even beginners. In such a case, there is no choice but to create your own function using the keyword `def` (right, derived from **define**). Let's have a look at the syntax:

```
def function_name(parameter1, parameter2, ...):
    # function's body
    ...
    return "return value"
```

After `def`, we write the name of our function (so as to invoke it later) and the names of **parameters**, which our function can accept, enclosed in parentheses. Do not miss the colon at the end of the line. The names of a function and its parameters follow the same convention as **variable names**, that is, they should be written in **lowercase with underscores between words**.

An indent of 4 spaces shows the interpreter where the function's body starts and where it ends. All statements in the function's body must be indented. You can make calculations inside your function and use the `return` keyword to send the result back. Only when the indentation is absent, the definition of the function ends.

Later, the parameters take on values passed in a function call. Those values we pass to a function are known as **arguments**. The only distinction between parameters and arguments is that we introduce parameters in a function definition and give arguments (some specific values) in a function call. Here is a bit less abstract example of a function:

```
# Function definition
def multiply(x, y):
    return x * y


# Function calls
a = multiply(3, 5)   # 15
b = multiply(a, 10)  # 150
```

In case you don't want to pass any arguments, the round brackets remain empty:

```
def welcome():
    print("Hello, people!")
```

You can also declare a sort of empty function with `pass` statement:

```
# This function does nothing (yet)
def lazy_func(param):
    pass
```

When you choose to call `lazy_func()` with an arbitrary value as its argument, nothing will happen. So `pass` is just a placeholder, but at least your code will be valid with it.

§1. Parameters vs arguments

It's not quite clear right now, what the parameters are, is it? In fact, parameters are just aliases for values, which can be passed to a function. Consider the following example:

Current topic:
✓ Declaring a function 10★ Stage 1

Topic depends on:
✓ Integer arithmetic 17★ Stage 1
✓ Invoking a function 16★ Stage 1

Topic is required for:
✓ Scopes 5★ Stage 1
✓ Arguments 7★ Stage 1
Function decorators
✓ Identity testing
✓ Lambda functions
✓ Functional decomposition 7★ Stage 1
✓ Search in a string 4★
Experiments with Python shell Stage 2
✓ Custom generators
Socket module
Requests: retrieving data
Testing user input
How to read a traceback
Series
Intro to NumPy
✓ Operations with tuple

```
def send_postcard(address, message):
    print("Sending a postcard to", address)
    print("With the message:", message)


send_postcard("Hilton, 97", "Hello, bro!")
# Sending a postcard to Hilton, 97
# With the message: Hello, bro!


send_postcard("Piccadilly, London", "Hi, London!")

# Sending a postcard to Piccadilly, London

# With the message: Hi, London!
```

As you can see, this function is a reusable piece of code, that can be executed with different arguments, i.e. different values passed into this function. Here, `address` and `message` are just the aliases under which the function receives values and then processes them in the body.

This function takes exactly 2 arguments, so you will not be able to execute it with more or less than 2 arguments:

```
send_postcard("Big Ben, London")


TypeError: send_postcard() missing 1 required positional argument: 'message'
```

§2. Execution and return

Our previous function only performed some actions, but it didn't have any **return value**. However, you might want to calculate something in a function and return the result at some point. Check the following example:

```
def celsius_to_fahrenheit(temps_c):
    temps_f = temps_c * 9 / 5 + 32
    return round(temps_f, 2)


# Convert the boiling point of water
water_bp = celsius_to_fahrenheit(100)
print(water_bp)  # 212.0
```

The keyword `return` is used to indicate what values the function outputs. Basically, it is the result of the function call. So, in the example above, we've stored the value returned by our function in the variable `water_bp`. Just to be sure, we printed the result.

One more thing to say is that functions do not necessarily have return values. The well-known `print()` function does not, in fact, return anything. Examine the code below:

```
chant = print("We Will Rock You")
print(chant)
```

And its output:

```
We Will Rock You
None
```

We declared the variable `chant` and invoked `print()`. Obviously, the function was executed. But the variable itself turned out to be the **None** object, which means the called function had nothing to return. The value of `chant` is **None**.

Python interpreter stops performing the function after `return`. But what if the function body contains **more** than one return statement? Then the execution will end after the first one. Please, keep that in mind!

§3. Conclusion

Thus, we've learned the syntax for declaring functions. Now you also know that:

- Parameters of a function are simply aliases, or placeholders, for values that you will pass to them. Parameters are re-initialized every time you call the function. Inside the function, you have access to these values, which means you can perform calculations on them.
- A function can simply perform an action without returning anything or return a specific result. If your function doesn't return anything, assigning its result to a variable or printing it will give you `None`.

Declaring your own functions makes your code more structured and reusable. Whenever you use the same piece of code more than once, try to create a function of it!

Report a typo

Thanks for your feedback!

Write here how we could improve this theory

Start practicing

Comments (16) Hints (1) Useful links (1) Show discussion