Function Arguments

by Sophia

WHAT'S COVERED

In this lesson, you will learn about creating functions with the use of arguments. Specifically, this lesson covers:

1. Adding New Functions
2. Function Arguments

1. Adding New Functions

For most of what we've worked with so far, we've used existing built-in functions that Python offers. As a reminder, in the context of programming, functions are a section of code that runs when it is called. We have the ability to pass data into those functions through parameters.

EXAMPLE

Here is an example of one of Python's built-in functions.

```
type(32)
```

Remember the **type() function**? It takes a value or a variable and returns its type. In our example, if we try to run it, it looks like it doesn't do anything, as the `type()` function takes the input, determines what the data type is, and returns it. If we would like to see what the `type()` function returned, we can output it to the screen by taking the `type()` function and passing the results into a `print()` function like this.

EXAMPLE

```
type(32)
print(type(32))
```

The output shows `int` type.

```
<class 'int'>
```

As expected, the type of 32 is an integer.

So, using this example function, the name of the function is `type()`. The expression in parentheses (32) of this function is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the `type()` function, is the type of that argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the return value. If a function call is used as an expression, the return value is the value of the expression. We will get into the return value in more detail in the next lesson.

What about new functions?

It is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We will discuss how a function is called later in this lesson.

Once we define a function, we can reuse the function over and over throughout our program. Here is an example:

EXAMPLE

```
def print_stuff():
    print("########################")
    print('Using for comment block')
    print("########################")
```

The keyword `def` indicates that this is a function definition. We are telling Python that we are defining a new function with this keyword. The name of our example function is `print_stuff()`.

KEY CONCEPT

The rules for function naming are the same as for variable naming: letters, numbers, and some punctuation marks are legal, but the first character can't be a number. You can't use a reserved keyword as the name of a function, and you should avoid having an existing variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions with parameters that take arguments as their inputs.

The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented. The body can contain any number of statements.

When we define a function, Python creates a variable with the same name. This is why you can't have the function name be the same as an existing variable prior to defining it. Look at the example and output below.

EXAMPLE

```
def print_stuff():
    print("########################")
```

```
    print('Using for comment block')
    print("########################")

print(type(print_stuff))
```

The output:

```
<class 'function'>
```

If we check the type of the `print_stuff()` function, we can see that type “function” is returned.

Calling a function

The syntax that we use to call a new function is the same as for built-in functions.

KEY CONCEPT

When we call a function, we're executing it. We can pass an argument (or not) to a function.

In the example below, we call the function by writing `print_stuff()` at the bottom of the code snippet.

EXAMPLE

```
def print_stuff():
    print("########################")
    print('Using for comment block')
    print("########################")

print_stuff() #here is where we are calling our print_stuff() function
```

Here is the expected output.

```
########################
Using for comment block
########################
```

TRY IT

Directions: Go ahead and try adding the `print_stuff()` function example above into the IDE and run it. Try changing the statement lines. Remember the colon and indentation.

It may not be clear why it is worth the trouble to divide a program into functions. Here are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place (the function itself).

- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Part of the skill of creating and using functions is to have a function properly capture an idea or task such as "find the smallest value in a list of values." Then you can reuse this function any time you need to repeat that task.

DID YOU KNOW

We have only covered a few functions so far in this course, but did you know there are over 70 built-in functions in Python? To learn more about what they are and their functions, check out docs.python.org/3/library/functions.html.

TERMS TO KNOW

Function Definition
A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.

def
The reserved keyword `def` indicates that you are defining a function.

---

2. Function Arguments

Some of the built-in functions we have seen require arguments. For example, the `print()` function takes in arguments to output them to the screen. Or, when you call the `math.sin()` function, you pass a number as an argument. The `math.sin()` function returns the sine of a number as a numeric value between - 1 and 1 which represents the sine of the angle given in radians.

Note: We will be using the `math.sin()` and `math.pow()` functions as examples in this section. You do not need to know specifically what these functions are and what they are used for per se; we are using them only in an example capacity. Also, in order to use these mathematical functions, we need to use the import system to access modules like `math`. The syntax `import math` imports the `math` module and allows us to perform special mathematical tasks on numbers. We will cover the import system later in Unit 3.

Some more notes on these math functions:

In the case of the `math.sin()` function, it takes one argument (only x) and it supports floats. That means if you did place an integer in as the argument, it will turn it into a `float`. So if you entered 2, it would convert it to 2.0.

So for example

```
import math
```

```
print(math.sin(0))
```

The output shows it as a float:

```
0.0
```

Some functions can take more than one argument; the `math.pow()` (power) takes two: the base and the exponent.

```
import math
print(math.pow(3))
```

The output with one argument:

```
Traceback (most recent call last):
  File "/home/main.py", line 2, in <module>
    print(math.pow(3))
TypeError: pow expected 2 arguments, got 1
```

Let's give the function what it needs (both arguments)

```
import math
print(math.pow(3,3))
```

This output looks better:

```
27.0
```

User defined functions

When we define our own function, arguments are assigned to variables called parameters inside the function.

Here is an example of a user-defined function that takes an argument. In the example, there is no call to the function yet, but the function was defined with one parameter `comment` that will accept an argument.

EXAMPLE

```
def print_stuff(comment):
    print("########################")
    print(comment)
    print("########################")
```

When called, this function assigns the argument to a parameter named `comment`. It will also print the value of `comment` in between the other two lines.

In the next example, we are calling the user-defined function by adding the last line. We added the string “My comment section” as the argument in the call.

EXAMPLE

```
def print_stuff(comment):
    print("########################")
    print(comment)
    print("########################")

print_stuff("My comment section") #here is where we are calling our print_stuff() function
```

So, the output looks like this.

```
########################
My comment section
########################
```

This function works with any value that can be printed. The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument (a string, the results of a calculation, an int, etc.).

See a variety of expressions as arguments in this example.

EXAMPLE

```
import math
def print_stuff(comment):
    print("########################")
    print(comment)
    print("########################")

print_stuff("My comment section")
print_stuff("Spam "*4)
print_stuff(1234)
print_stuff(math.pi)
```

The output shows:

```
########################
My comment section
########################
########################
Spam Spam Spam Spam
########################
########################
1234
########################
```

```
######################
3.141592653589793
######################
```

The argument is evaluated before the function is called, so in the examples, the expressions `'Spam '*4` are only evaluated once to output “Spam” 4 times.

TRY IT

Directions: Try adding this code to the IDE and try some different expressions as arguments.

You can also use a variable as an argument.

EXAMPLE

```
def print_stuff(comment):
    print("######################")
    print(comment)
    print("######################")

myVar = "My comment section"
print_stuff(myVar)
```

The output shows the commented section.

```
######################
My comment section
######################
```

The name of the variable we pass as an argument `myVar` has nothing to do with the name of the parameter `comment`. Here in the `print_stuff()` function, the value is referenced as the `comment` parameter.

We can also add multiple parameters as part of the function definition statement to use in different ways. Each parameter is separated by a comma. Let’s look at an example of a function that takes in two numbers and displays the mathematical equation to add them to the result.

EXAMPLE

```
def display_add_two_num(val1, val2):
    print(val1,"+",val2, "=", val1+val2)

display_add_two_num(2, 4)
display_add_two_num(3, 8)
```

The output displayed:

```
2 + 4 = 6
```

```
3 + 8 = 11
```

Each time we call the function, we have to simply pass in the arguments into the function and change the argument values. You can have as many parameters for a function as you need, like this example that now takes in 4 numbers (`val1 - val4`).

EXAMPLE

```
def display_add_four_num(val1, val2, val3, val4):
    print(val1,"+",val2,"+",val3,"+",val4, "=", val1+val2+val3+val4)

display_add_four_num(2, 4, 3, 8)
```

The output with the sum of 17.

```
2 + 4 + 3 + 8 = 17
```

BIG IDEA

We have been using the terms “parameters” and “arguments” quite a bit in this lesson. Before we finish, let’s refresh on what they mean and how functions are created and utilized. Let’s break down this example line by line:

```
1 def myOwnFunction(parameter1, parameter2):
2     print(parameter1, "+",parameter2, "=", parameter1 + parameter2)
3
4 myOwnFunction(2,4)
5 myOwnFunction(3,8)
6
```

Function definition

- Line 1 Header: Functions are created (defined) using the `def` keyword followed by the name of the function. Remember to use proper naming techniques. Our example function is called `myOwnFunction`. It is followed by the parentheses that contain any amount of parameters you need for the task (can be none to however many). Remember to use a comma between parameters. Our example has two parameters called `parameter1` and `parameter2`. You finish off this function’s header with a colon.
- Line 2 Body: These are the statements of the functions. There can be any number of statements. They need to be indented. This function is printing out each of the arguments passed and the total when added together.

Function call

- Line 4: 1st call to the `myOwnFunction()` function with arguments of integers `2` and `4`. Each argument is separated by a comma. Notice that this is not indented as it is not part of the function.
- Line 5: 2nd call to the `myOwnFunction()` function with arguments of integers `3` and `8`.

Each time the `myOwnFunction()` function is called, it accepts two passed arguments that are assigned to the parameters of the function. These parameters are used inside the function.

SUMMARY

In this lesson, we learned that we can **add new functions** instead of using Python's built-in functions. We discussed how to set the function definition by including the header and the body. We also discovered that when we define a function, Python creates a variable with the same name. We also saw that the IDE can provide tips when writing code, in the examples with the `math.sin()` and `math.pow()` functions. Finally, we learned that we can pass **arguments into our functions** and that we can have as many arguments in a function as we need.

Best of luck in your learning!

TERMS TO KNOW

Function Definition

A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.

def

The reserved keyword def indicates that you are defining a function.