Simplifying Your Code: Functions and How to Use Them

Introduction to Python Programming

# Overview

In this module, you are going to learn how to write functions. The ability to write functions is a huge step up in programming ability, and programmers who learn this skill will be able to write more organized and efficient code. Your code will be easier to maintain, understand, and share with others or share between your own scripts. Similarly, you can define your own functions. These can be used anywhere in your code, so that you don’t need to retype the content of that function every time you want to use it. In short, functions allow programmers to write a piece of code one time and use it everywhere, many times.

Unfortunately, many people who pick up programming only learn to write scripts. They never really learn how to write their own functions, and this often leads to very long and confusing code. Functions also help programmers stick to one of the universal laws of programming captured in the acronym DRY— that is, “Don’t repeat yourself!” This principle can also be understood by its opposite term WET, which can mean “Write everything twice,” “Waste everyone’s time,” or “We enjoy typing.” Thus, the goal in using Python, or any language, is to be effective, yet succinct.

## Example

As we begin to discuss functions in greater detail, let’s start with a simple example of a programming scenario. Remember our *if/else* statement for determining the shipping cost of a package from the previous module? This was our initial code:

|  |
| --- |
| **if** package\_weight < 10:  shipping cost = 5 **elif** package\_weight < 15:  shipping\_cost = 7 **else**:  shipping\_cost = 10 |

Let’s imagine that the code we initially used to run our shipping company has now grown to a much more feature-rich and complex system. We now keep track of package pricing, inventory, shipping routes, etc., and our code base is now many thousands of lines. In addition, our code needs to calculate package shipping costs at five different locations in the code.

To attempt to solve this issue, we decide to copy the code above and paste it into these five different places. However, this is not a good choice! Why not? This is because a month later we need to change the highest shipping cost to $11 instead of $10. Now, we need to search through our code to find all five of those places and change the cost from $10 to $11 in each of them. Unfortunately, we forgot that we had the code in five locations, and we only changed four of them, inadvertently creating various price structures.

On reflection, we can recognize these inconsistencies in our code and also realize that it took too much time to search through our code to make the changes. Yet, how could we have avoided this type of situation in the first place? By using a function! Use of a function in this type of scenario can give us a way to package the code, so we can simply refer to the entire code block as one unit by name.

Next, let’s consider how we might go about writing functions in Python.

# Writing Functions

Here is the syntax for our example of a Python function:

|  |
| --- |
| **def** **calculate\_shipping\_cost**(package\_weight):  **if** package\_weight < 10:  shipping cost = 5  **elif** package\_weight < 15:  shipping\_cost = 7  **else**:  shipping\_cost = 10  **return** shipping\_cost |

Notice that the overall syntax is similar to the *if/else* statements and *for* loops which we have previously considered, and the line at the top also ends in a colon or ‘:’ symbol. In this syntax, the first line is called the *function signature*. The indented code below is the *content of the function*, ending with a *return line*. Let’s examine each of these in more detail.

1. **Function signature**: The first line starts with ‘**def**’ which is the keyword that lets Python know that we are *defining a function*. The next part, ‘**calculate\_shipping\_cost**‘, is the name of the function. This name can be anything we want as long as it starts with a letter or underscore and only contains letters, underscores, or numbers. We then indicate how many inputs the function will take. This is determined by the number of argument names that are listed between the parentheses. In the example above, our function only takes one argument, and this can be seen because there is only one name, ‘package\_weight’, in between the parentheses. Thus, ‘package\_weight’ is the name of this argument *inside* of the function. Finally, the first line ends in a colon or ‘:’ symbol.
2. **Function content**: Below the function signature is the *content of the function.* These lines of codes are indented and run every time the function is used (i.e., executed).
3. **Return line**: The last line is *the return line*. We start this line with the ‘**return**‘ keyword, and then write the variable that we want to return, which in this case is ‘shipping\_cost’.

Defining the function like this allows us to reuse it to process various package weights. Now, let’s consider an example.

## Example

In the code below, we define our function and then use the function three times for three different package weights (5, 21, 14).

|  |
| --- |
| **def** **calculate\_shipping\_cost**(package\_weight):  **if** package\_weight < 10:  shipping cost = 5  **elif** package\_weight < 15:  shipping\_cost = 7  **else**:  shipping\_cost = 10  **return** shipping\_cost  ship\_cost\_1 = calculate\_shipping\_cost(5)  ship\_cost\_2 = calculate\_shipping\_cost(21)  ship\_cost\_3 = calculate\_shipping\_cost(14)  print(ship\_cost\_1)  print(ship\_cost\_2)  print(ship\_cost\_3) |

*Note*: Only the indented lines are part of the function, and the function ends at the last indented line, which is also the return line. The lines that follow are not indented.

Every time we use the function, as we do in the line “ship\_cost\_1 = calculate\_shipping\_cost(5)”, all of the code inside the function runs with whatever value is passed to it. So, in this case we pass to the function the value of 5. This value then runs the code inside the function with the ‘package\_weight’ argument equal to the value 5. The function calculates the shipping cost and returns that value from the function. This is the output of the function, and this is the value that is assigned to ‘ship\_cost\_1’

The above example is very simple. We have defined our function and then used it immediately after we have defined it. However, this is not usually the case. Usually, function definitions are at the bottom of the file and then are used throughout the code. Functions can also be defined in a separate file and imported into the script and used. The structure of a project can be even more involved than that with multiple files of code, but this consideration is beyond the scope of this course.

# Built-In Functions

We have already seen several examples of built-in functions in this course, but now that we have an understanding of what functions are, maybe we can appreciate them a little more. *Built-in functions* are functions that are built into Python and can be used at any time in creating code. Thus far, we have seen the functions *type* and *print*, but a complete list of all built-in functions can be found [here](https://docs.python.org/3/library/functions.html).

Let’s now consider the round function, another example of a built-in function in Python.

## The Round Function

The round function is a built-in function that can be used without importing anything special. We can use the round function anytime we need to round a number, and we don’t have to type out all of the lines of code that go into rounding a number. There are actually several lines of code running each time we use the function *round*. However, we don’t see them because they have been packaged up into the round function. (As a side note, the underlying lines of code that define *round* are not lines of Python code. Rather the code is written in *C* so the function is faster.)

The round function can be used in this way:

|  |
| --- |
| my\_rounded\_num = round(1.7) print(my\_rounded\_num) |

## Other Examples

Overall, using built-in functions is very simple, and here are some other examples:

|  |
| --- |
| my\_name = "William" print(my\_name) a\_number = 17.5 b\_number = round(a\_number) print(a\_number) |

Now, let’s wrap up this week’s module with a look at what is called the standard library.

# The Standard Library

The *standard library* is a library of Python modules that provide many additional functions, well beyond those that are included in the built-in functions. The standard library was developed by the Python developers and almost always comes with Python. These modules provide essential functions that many other modules are built on. You can see a list of the modules included in the standard library [here](https://docs.python.org/3/library/).

In a future lesson, we will dig into a few of these, but for now let’s look at how to import a Python module from the standard library so it can be used in your code. Unlike built-in functions, which can be used anytime, functions in the standard library can only be used after the module has been imported into your code.

So, let’s say we want to use the ‘*os*’ module. This is very simple! We can import it like so:

|  |
| --- |
| **import** os |

Important statements almost always go at the very top of your code, and in a Jupyter notebook they should be in the first cell. You must run this cell to import the module. Once you run the cell, that module will be available anywhere in your notebook.

Now, try the following in the first cell in a notebook:

|  |
| --- |
| **import** os os.getpid() |

This imports the *os* module and then uses the *getpid* function that is inside that module. This just returns the process ID of the Python process (in this case, the Jupyter Notebook).

We don’t need to be an expert in this function, and this is only an example of how to use a function from the *os* module. You can learn more about the *os* module [here](https://docs.python.org/3/library/os.html).

We can also import just the function names, and not the module name, in this way:

|  |
| --- |
| **from** os **import** getpid() getpid() |

It may seem like this code is more efficient since it looks like we are only importing one function and not the entire module, but this is not correct. In both cases, the entire module is being read into memory. The only difference is that in one example you would write ‘os.getpid()’ to use the *getpid* function, and in the other example you need to write ‘getpid()‘.

As mentioned, in a future lesson we will look at a few more examples of modules included in the standard library.

# Conclusion

In this week’s lesson, we’ve considered the importance of learning how to write your own functions. Learning this important programming skill will lead to more organized and efficient code which will also be easier to understand, maintain, and share. Python’s built-in functions and the standard library provide a good foundation for using functions.