Lesson 5: Modular Programming With Functions

Introduction

Up to this point, all of your programs have been executing statements in a list, and the programs have been small, which hasn't given us much of a need for grouping and organizing our code. However, as your programs become more complex, organization will become key in keeping your code manageable. Programmers call grouping and organizing code modular programming. And this is where functions come in.

You can think of a function as a way to name and group a set of Python statements. Then, any time you want to run those statements, you can just call on that function and they'll run.

In this lesson, I'll show you how to define, pass values to, and call functions in your Python programs. This will make your life as a programmer easier because you'll be able to take large problems and break them into smaller, more manageable pieces.

So when you're ready to begin, let's head over to Chapter 2, and we can start looking at the basics of Python functions.

Python Functions

I think the best way to start discussing about functions is to stop and look at some of the functions we've already worked with. For example, input. Take a look below:

```
age = eval(input("Enter your age: "))
```

You've been using this function for a few chapters now. I even introduced it to you as a function, but I didn't mention much more about it. Now's the perfect time for us to look a little deeper.

The creators of Python figured that many programmers would want some way to get input from their users. For that reason, they created the function called input. It's input's job to get user information and store it in whatever variable is on the left side of the equal sign.

The input function contains all of the code that's needed to perform this action. We never had to worry about the code because it was all contained inside the function Python provided. And that is the beauty of functions. It saved us time and effort and allowed us to concentrate on more important things—like the rest of our code! And it gets better. We can create our own functions. We can write a set of code, save it in a function, and then any time we want to run that code, we just call the function.

So, let's start with the general form of a Python function. Basically it should look like this:

```
def < function name > ( < parameter list >):
    < documentation string >
    < the function's code >
```

Notice the use of the keyword *def*. You can think of this as shorthand for define. After all, we're defining the function, right?

The next thing I want to point out is the parentheses and the parameter list. Be aware that you're required to have those parentheses there, even if your function doesn't have any parameters in its *parameter list*. The parameter list is just a list of the variables that this function needs from the calling statement in order to work correctly.

The idea is that the line of code that calls a function will have a list of values. These values in the calling line of code are known as *arguments* to the function. The values of those arguments will be stored in the variables in your function's parameter list.

I also want to point out a couple of things that should remind you of the decisions and loops you've been writing. Notice the colon at the end of the first line and how the statements inside the body of our function are indented. Just like the other programming structures, you're allowed to include as many statements inside your function as you like. But, you need to indent the code so that Python knows which statements are a part of the function and which aren't.

Additionally, although it isn't required, it's strongly recommended that you include at least one *documentation string* or *docstring* as the first line of your Python functions. This *string literal* is just a single line enclosed in three sets of double quotes that gives the purpose of the function. Although it doesn't affect the functionality of your code, there are tools that will go through your programs and use these strings to generate documentation for your programs. This helps others (who may want to use your code) have a clear understanding of how to use it.

One final thing to note about function definitions, you'll need to be sure to define your function before placing the line of code in the file that calls it. Typically, programmers will define all of their functions at the beginning of a file and then have all of the calling statements together at the end of the file.

With this general form in mind, let's get down to business and create a function of our own. We'll start with a small function that displays a welcome message. Let's say you're writing multiple gaming programs and you want to print a list of the rules when each program first starts. Sure, you could put the code into the main part of your program, but we're going to organize our code better so that all of the introductory welcoming is done within a single function.

Why don't you give it a shot before you move down to the next set of lines to see how it's done? Of course, your message will probably be different than mine, but that's okay. The goal here is to be able to write the function:

```
def < function name > ( < parameter list >):
    < documentation string >
    < the function's code >
```

See my code

If you used my code, did you try to type this in as a separate program, save the file, and then run it? Or maybe you decided to define this function right in the IDLE command line interpreter. If so, then you noticed that nothing was printed on the screen. Well, that's because you just defined the function, you didn't call it.

Calling a function is simply writing a line of code that uses the name of the function and any arguments that you want stored in the parameter list. For this welcome function, that would be something like this:

```
print_welcome()
```

Be aware, once again, that the parentheses are needed, even though we aren't passing any values to the function. Go ahead and either type this call into the interpreter directly or place the code in your file and run it again.

Again, here's the really useful part of functions. Now that I've defined print_welcome, I can use those statements whenever I want simply by calling the function. For example, I might have a program file that does something like this:

```
def print_welcome():
    """This function prints two lines of text"""
    print ("Welcome to my program")
    print ("I hope you like it")


print_welcome()
print()
print ("To say it again")
print()
print_welcome()
```

Okay, now I'll admit that this might not be something you'd really want to do in a program. However, it does demonstrate my point. For example, maybe instead of having two output statements, maybe this function was 1000 lines of code that perform some complex calculation. Once the function is defined, I can run all 1000 lines of code just by placing a single line of code that calls the function. Pretty useful, don't you think?

These are the basics for writing and calling functions. Now I want to move ahead and discuss about how we can pass values to functions. We'll do that in Chapter 3.

Passing Values to Functions

You probably noticed our little example in the previous chapter didn't receive any values or return any values. Instead, it did some printing. Other programming languages like to call functions like this *procedures* or *subroutines* and reserve the name *functions* for things that return values.

Well, Python skips over this and just labels everything as a function. Technically, every Python function does return a value. We'll discuss about this a little later in this lesson.

First, let's learn how to pass values into a function. Remember, we can make a list of parameters inside the parentheses. And since Python doesn't require us to use data types, that's where we put the variable names that we want to use. If you happen to have more than one parameter in your list, just separate each one with a comma.

For example, maybe you want to write a function that will print whatever value is passed to it. This function might look something like this:

```
def print_value(value):
    """This function prints the value passed in"""
    print ("Your value is:", value)
```

The interesting thing about writing a function like this in Python is that when you call print_value, it doesn't matter whether you pass it a string or a number. Either way, the function will work and print whatever value you pass it.

Try this out for yourself by attempting to call this function with the following statements:

```
print_value(5)
print_value("number")


name = "Pat"
print_value(name)
```

Notice that the first call passes the numeric value 5 to the function, the second passes a string literal, *number*, and the third passes a variable.

Sometimes when folks just begin writing functions, they get a bit confused by the variables. I want to take a second and look a little deeper at my third example. Again, Python doesn't care whether you're giving it a number, string, or variable. The function runs fine no matter what.

However, some interesting things happen when you pass variables. First, you'll notice that we have one variable name in the calling statement and a second variable name in the function's parameter list. This is okay because we're passing the value, Pat, stored in name, which is then stored in the variable value inside the function.

Here's another interesting point. You might be wondering what might happen if you attempt to change the value inside the function. And what would happen outside the function call? Let's write a new function that makes such a change, and then call on it to see exactly what's happening:

```
def change_value(value):
    """This function changes the value passed in to 1"""
    print ("Inside, value is:", value)
    value = 1
    print ("Inside, value is changed to:", value)


number = 5
print ("Outside, number is:", number)
change_value(number)
print ("Outside, number is now:", number)
```

The idea with this program is that we start our number variable with the value 5. We print it to make sure that's what it is, then call change_value. This function has an output statement to confirm what value it's storing, then changes it and prints the new value. Since we're at the end of the function, control transfers back to the place where the function was originally called and the next statement is run.

If you try this out, you'll see that in fact the final print statement displays the value 5. Again, if you think about passing values the way I described it earlier (we copy the value of the variable into the new variable inside the function) then it should make sense. That's because we're modifying the variable that is inside the function and never touching the variable that is outside.

But there's something else that may have occurred to you here. You might be wondering what would happen if I changed that final print statement so that it tries to print the value of the value variable, instead of the number variable.

Well, if you tried to do this, you'd find that this code wouldn't run. That's because of the variable's *scope*. Essentially, the scope of a variable is where the variable is active in the program.

When you list a variable in a function's parameter list, you're creating a local variable. This is a variable that can only be used inside the function. So, when you leave the function, this variable will actually be destroyed, and you cannot access it anymore.

This ability for the functions to make a copy of the value and store it in a local variable while the function is running is generally seen as a good thing. After all, typically, you don't want to have to worry about passing a variable over to a function and not know whether the function is changing the value or not. But what happens if this is exactly what you want? That is, what happens if you want to have a function modify the contents of a variable that is not part of the parameter list?

For that, you can use the *global* statement. This statement tells Python that you're going to want to access and be able to modify a variable that isn't part of the current function. To use the global statement, just put the keyword global, followed by the list of variables that you want to access. The following example uses the global statement:

```
def change_number():
    """This function changes the value passed in to 1 (global)"""
    global number
    number = 1


number = 5
print ("Outside, number is:", number)
change_number()
print ("Outside, number is now:", number)
```

Now when the last print statement is executed, the number 1 is printed. That's because our function requests global access to the number variable. Then, when it makes the change to number inside the function, the changes are felt outside the function as well.

To prove this to yourself, go ahead and place a # in front of the global statement in the function and run the code again. This time, you'll find that 5 is printed both times.

So you've learned how to write these functions and pass values, but what about getting values back? After all, the input function that we started the lesson with does return a value that we can store in a variable, right? Well, I've been saving that one for our next chapter, along with some other interesting facts about functions.

Returning Values From Functions

Passing a value back or returning a value from a function is done with the keyword *return*. You'll just place this keyword, followed by the value you want returned, at the end of your function. It is very important to note that once Python reaches the return statement, control will pass back to the statement that called the function. This means that any code you put after the return statement will never be executed.

Let's try this by practicing everything we've learned about functions so far, including the return statement. We'll write a function that will compute the square of a number. Think back to math class. The square of a number is just a number multiplied by itself. So, this function might look something like this:

```
def square(num):
    """This function calculates the square of a number"""
    result = num * num
    return result
```

Here, my last line in the function is the keyword *return* followed by the variable *result*. This is going to return to the calling statement whatever value is stored in result.

Maybe as you look at this code, you're wondering if it's possible to skip the first statement and just do this:

```
return num * num
```

The answer is yes. In fact, many people prefer to write functions that do simple calculations like this. Either way is fine with me. I'm just showing you a variety of ways to get the job done.

Let's practice calling this function by using the loops that we learned from the last lesson. Write some code that will call the square function for each of the numbers between 1 and 10 and print each of the results:

```
for i in range(1,11):
    print (square(i))
```

One thing to notice with the code above is that we're calling the square function along with the print statement. This is going to print the value that is returned by square. You'll always need to remember that when you call a function you're going to want to do something with the result that comes back. Of course, the code will still run if you leave out the print part in the loop above, but you won't see anything on the screen.

This leads me back to something I said earlier in this lesson when I discussed about the differences between functions that return values and those that don't. Remember how I said that technically all Python functions return a value? Well, that's because at the end of every function, there's an implied statement that says *return None*. You can confirm this by simply calling on one of the functions from a previous chapter and printing its result. For example, try out the following code:

```
def print_welcome():
    """This function prints two lines of text"""
    print ("Welcome to my program")
    print ("I hope you like it")


print (print_welcome())
```

Running this program will give you the two lines of output that you're expecting, followed by the return value of this function, which is *None*. None is a special piece of data in Python. Essentially it's just an empty placeholder. This is similar to null in other languages. At any rate, now you see why technically all Python functions return a value.

By the way, are you wondering why we didn't see the None when we called our square function? Remember, when the return statement is reached, Python transfers control back to the statement that called the function. For this reason, we'll never see the None for our functions that return a value.

Now that you have the basics of Python functions, I want to tell you about a few extras.

Default Arguments

There may be times when you want to allow your function to be called with a fewer number of arguments than is required in a parameter list. This is called a *default argument*. In this case, you need to provide a *default value* in your parameter list, which is done by placing an equal sign and a value next to the variable in your parameter list.

For example, let's say you want to allow the square function to be called without having to specify a number. If this happens, you just want square to calculate the square of 1. You can do this by doing the following:

```
def square(num = 1):
    """This function calculates the square of a number"""
    result = num * num
    return result
```

This allows me to call square with the following statement:

```
print (square( ))
```

The purpose of a default value may not be clear based on this example. But, there are times when it may be quite helpful. For example, maybe you want a function that will print a prompt and then allow the user a specified number of chances to enter either yes or no. You could use default arguments to do something like this:

```
def prompt_user(prompt, num_tries = 2):
    """This function prompts the user a certain number of times"""
    answer = input(prompt)

    while (answer != "yes" and answer != "no" and num_tries > 1):
       num_tries = num_tries - 1
       print ("Try again")
       answer = input(prompt)
```

Now that you've given a default value for num_tries, you're able to call this function with either of the following statements:

```
prompt_user("Enter yes or no: ")
prompt_user("Enter yes or no: ", 4)
```

Keyword Arguments

The last thing I want to show you about functions is the use of *keyword arguments*. You can use keyword arguments to specify which variables in the function's parameter list should be storing which values. This is done in the statement that calls the function. For example, using the previous prompt_user function, we might use keyword arguments and call it with the following:

```
prompt_user(prompt="Hello")
prompt_user(num_tries=1, prompt="Hi")
```

This allows us to specifically assign the function's variable to a value. It also has a side effect. You're allowed to call the functions with the parameters out of order, as is shown in the second function call.

The way the parameter list is set up in prompt_user, the calling statement should list the prompt first, followed by the number of tries. However, since we're specifically assigning the values to the function's variables, we're allowed to do them in whatever order we want. Pretty cool, don't you think?

Summary

Congratulations! With this lesson under your belt, you have what you need to start writing Python functions. In this lesson, you learned how to define and call on your own functions. You also found out how to pass data back and forth between function calls and function definitions. You even learned some pretty involved ways of passing this data.

While using functions in your programs does require some thought and practice, keep at it because I think you'll find as time goes on that functions are indispensable. They not only help to make for less coding, but they also help better organize your programs and make them more modular.

Now that I mention of modular code, in the next lesson, we're going to learn more about modules in Python. So far we've been writing all of our code in a single module. However, Python allows us to create multiple modules to better organize the code in our programs. We'll take a look at some common modules that have already been written for us and then move on and write some of our own.