Chapter 10 - Functions

A function is a structure that you define. You get to decide if they have arguments or not. You can add keyword arguments and default arguments too. A function is a block of code that starts with the **def** keyword, a name for the function and a colon. Here's a simple example:

```
>>> def a_function():
        print("You just created a function!")
```

This function doesn't do anything except print out some text. To call a function, you need to type out the name of the function followed by an open and close parentheses:

```
>>> a_function()
You just created a function!
```

Simple, eh?

An Empty Function (the stub)

Sometimes when you are writing out some code, you just want to write the function definitions without putting any code in them. I've done this as kind of an outline. It helps you to see how your application is going to be laid out. Here's an example:

```
>>> def empty_function():
        pass
```

Here's something new: the **pass** statement. It is basically a null operation, which means that when **pass** is executed, nothing happens.

Passing Arguments to a Function

Now we're ready to learn about how to create a function that can accept arguments and also learn how to pass said arguments to the function. Let's create a simple function that can add two numbers together:

```
>>> def add(a, b):
        return a + b

>>> add(1, 2)
3
```

All functions return something. If you don't tell it to return something, then it will return None. In this case, we tell it to return **a + b**. As you can see, we can call the function by passing in two values. If you don't pass enough or you pass too many arguments, then you'll get an error:

```
>>> add(1)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: add() takes exactly 2 arguments (1 given)
```

You can also call the function by passing the name of the arguments:

```
>>> add(a=2, b=3)
5
>>> total = add(b=4, a=5)
>>> print(total)
9
```

You'll notice that it doesn't matter what order you pass them to the function as long as they are named correctly. In the second example, you can see that we assign the result of the function to a variable named **total**. This is the usual way of calling a function as you'll want to do something with the result. You are probably wondering what would happen if we passed in arguments with the wrong names attached. Would it work? Let's find out:

```
>>> add(c=5, d=2)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: add() got an unexpected keyword argument 'c'
```

Whoops! We received an error. This means that we passed in a keyword argument that the function didn't recognize. Coincidentally, keyword arguments are our next topic!

Keyword Arguments

Functions can also accept keyword arguments! They can actually accept both regular arguments and keyword arguments. What this means is that you can specify which keywords are which and pass them in. You saw this behavior in a previous example.

```
>>> def keyword_function(a=1, b=2):
        return a+b

>>> keyword_function(b=4, a=5)
9
```

You could have also called this function without specifying the keywords. This function also demonstrates the concept of default arguments. How? Well, try calling the function without any arguments at all!

```
>>> keyword_function()
3
```

The function returned the number 3! Why? The reason is that **a** and **b** have default values of 1 and 2 respectively. Now let's create a function that has both a regular argument and a couple keyword arguments:

```
>>> def mixed_function(a, b=2, c=3):
        return a+b+c

>>> mixed_function(b=4, c=5)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: mixed_function() takes at least 1 argument (2 given)
>>> mixed_function(1, b=4, c=5)
10
>>> mixed_function(1)
6
```

There are 3 example cases in the above code. Let's go over each of them. In the first example, we try calling our function using just the keyword arguments. This will give us a confusing error. The Traceback says that our function accepts at least one argument, but that two were given. What's going on here? The fact is that the first argument is required because it's not set to anything, so if you only call the function with the keyword arguments, that causes an error.

For the second example, we call the mixed function with 3 values, naming two of them. This works and gives us the expected result, which was 1+4+5=10. The third example shows what happens if we only call the function by passing in just one value...the one that didn't have a default. This also works by taking the "1" and adding it to the two default values of "2" and "3" to get a result of "6"! Isn't that cool?

*args and **kwargs

You can also set up functions to accept any number of arguments or keyword arguments by using a special syntax. To get infinite arguments, use *args and for infinite keyword

arguments, use **kwargs. The “args” and “kwargs” words are not important. That’s just convention. You could have called them *bill and **ted and it would work the same way. The key here is in the number of asterisks.

*Note: in addition to the convention of *args and **kwargs, you will also see *a and **kw from time to time.*

Let’s take a look at a quick example:

```
>>> def many(*args, **kwargs):
        print(args)
        print(kwargs)

>>> many(1, 2, 3, name="Mike", job="programmer")
(1, 2, 3)
{'job': 'programmer', 'name': 'Mike'}
```

First we create our function using the new syntax and then we call it with three regular arguments and two keyword arguments. The function itself will print out both types of arguments. As you can see, the **args** parameter turns into a tuple and **kwargs** turns into a dictionary. You will see this type of coding used in the Python source and in many 3rd party Python packages.

A Note on Scope and Globals

Python has the concept of **scope** just like most programming languages. Scope will tell us when a variable is available to use and where. If we define the variables inside of a function, those variables can only be used inside that function. Once that function ends, they can no longer be used because they are **out of scope**. Let’s take a look at an example:

```
def function_a():
    a = 1
    b = 2
    return a+b

def function_b():
    c = 3
    return a+c

print( function_a() )
print( function_b() )
```

If you run this code, you will receive the following error:

```
NameError: global name 'a' is not defined
```

This is caused because the variable **a** is only defined in the first function and is not available

in the second. You can get around this by telling Python that **a** is a **global** variable. Let's take a look at how that's done:

```
def function_a():
    global a
    a = 1
    b = 2
    return a+b

def function_b():
    c = 3
    return a+c

print(function_a())
print(function_b())
```

This code will work because we told Python to make **a** global, which means that that variable is available everywhere in our program. This is usually a bad idea and not recommended. The reason it is not recommended is that it makes it difficult to tell when the variable is defined. Another problem is that when we define a global in one place, we may accidentally redefine its value in another which may cause logic errors later that are difficult to debug.

Coding Tips

One of the biggest problems that new programmers need to learn is the idea of "Don't Repeat Yourself (DRY)". This concept is that you should avoid writing the same code more than once. When you find yourself doing that, then you know that chunk of code should go into a function. One great reason to do this is that you will almost certainly need to change that piece of code again in the future and if it's in multiple places, then you will need to remember where all those locations are AND change them.

Copying and pasting the same chunk of code all over is an example of **spaghetti code**. Try to avoid this as much as possible. You will regret it at some point either because you'll be the one having to fix it or because you'll find someone else's code that you have to maintain with these sorts of problems.

Wrapping Up

You now have the foundational knowledge necessary to use functions effectively. You should practice creating some simple functions and try calling them in different ways. Once you've played around with functions a bit or you just think you thoroughly understand the concepts involved, you can turn to the next chapter on classes.