Debugging Functions

by Sophia

WHAT'S COVERED

In this lesson, you will learn about common errors specific to functions. Specifically, this lesson covers:

1. Common Function Errors
2. Function Argument Errors
3. Scope Errors

1. Common Function Errors

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**. Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program. Remember that statements inside the function are not executed until the function is called. A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function! Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

Take the following example:

EXAMPLE

```
def function_4(num):
    if num > 1:
        num = num - 2
    else:
```

```
        num = num + 2
    return num

def function_3(num):
    if num > 1:
        num = num / 2
    else:
        num = num / -2
    return num

def function_2(num):
    if num > 1:
        num = num * 2
        num = function_3(num)
    else:
        num = num * -2
        num = function_4(num)
    return num

def function_1(num):
    if num > 0:
        num += 1
        num = function_2(function_3(num))
    else:
        num -=1
        num = function_2(function_4(num))
    return num

print('output = ', function_1(5))
```

As we can see, reading it from top to bottom does not work. This is due to the fact that there are four functions that are simply being defined. What we must do in this case is trace the path of the function calls. The `print()` function is calling `function_1` and passing a value of 5. With that, `function_1` accepts that value as the variable `num` and checks num's value. Since 5 is greater than 0 in the if statement, the next statement is called to increase `num` by 1.

Next, the following statement is called.

```
num = function_2(function_3(num))
```

In this statement, there are two separate function calls. The call to `function_3` is called first, passing in `num` being 6 at this time. Once a value is returned from `function_3`, the returned value is passed to `function_2`. We could have written a similar set of statements that would do the same thing.

```
result = function_3(num)
num = function_2(result)
```

Let's step through the next part as the call goes to `function_3`.

```
def function_3(num):
    if num > 1:
        num = num / 2
    else:
        num = num / -2
    return num
```

Remember that `num` was set to 6 in `function_1`, so on the first check, `num` is greater than 1; therefore, the statement has `num` divided by 2 and set back to `num`. So now num's value is 3. Afterwards, `num` is being returned from `function_3`; it's immediately passed into `function_2`.

```
def function_2(num):
    if num > 1:
        num = num * 2
        num = function_3(num)
    else:
        num = num * -2
        num = function_4(num)
    return num
```

As `num` is equal to 3, on the check, `num` is > 1, so `num` is multiplied by 2 so `num` is equal to 6. Then `num` is passed to `function_3` where we do the check again if `num` > 1. It is, so `num` is divided by 2 and passed back. Afterwards, 3 is passed back again to `function_1` which is then passed back to the original `print()` function statement. This outputs to the screen that the output = 3.0.

Based on this example, when we read a program, we don't always want to read from top to bottom. Sometimes it makes more sense if we follow the flow of execution.

TRY IT

Directions: Try following this example in the IDE. Change the initial `print()` function's passed value and without running the program, follow the flow of execution and see if you can come up with the expected output. Then run the program and see if you are correct.

TERM TO KNOW

Flow of Execution

The order in which statements are executed is called the flow of execution. Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

2. Function Argument Errors

One of the more common issues with functions in Python is with the positions of the arguments that are being passed into a function. The order is important when we have multiple arguments being passed in, as the position of the arguments reflects what variable is assigned. Let's look at an example with multiple arguments passed.

EXAMPLE

```
def priceInformation(qty, item, price):
    print(f'{qty} {item} cost ${price:.2f}')

priceInformation(12, 'strawberries', 5.99)
```

Here is the output.

```
12 strawberries cost $5.99
```

Something to note with this example is that we're using the `print()` function with formatting incorporated into it. The price parameter is expecting a `float` and ensures that it is displayed with 2 decimal places.

KEY CONCEPT

Controlling the output through formatting.

When we output variables, there are times that we want to control the formatting of the output. To use formatted string literals, we start a string with the letter "f" before the opening quotation mark.

```
print(f'{qty} {item} cost ${price:.2f}')
```

Inside of the string, you can write a Python expression between the left curly bracket and the right curly bracket ({}) that refers to variables or literal values. In our example, we output the `qty` and `item` as we expect it to, but we've added formatting so that the `price` is limited to two decimal places (price:.2f). The colon (:) indicates the start of the formatting for the number of places after the decimal value.

```
print(f'{qty} {item} cost ${price:.2f}')
```

The first value that's passed in (12) is assigned to `qty`, the second value (strawberries) is assigned to `item`. The last variable is assigned to `price`.

TRY IT

Directions: Your turn, try the code example above in the IDE and see if you get the same output screen.

As programs get larger, it can be difficult to remember the order of the input fields and we can easily make mistakes when passing in arguments. Sometimes, if we have different data types, we may have an error display due to a type conversion, such as if we swapped the `item` and `price`.

EXAMPLE

```
def priceInformation(qty, item, price):
    print(f'{qty} {item} cost ${price:.2f}')

priceInformation(12, 5.99, 'strawberries')
```

The output flags an error message.

```
Traceback (most recent call last):
  File "/home/main.py", line 4, in <module>
    priceInformation(12, 5.99, 'strawberries')
  File "/home/main.py", line 2, in priceInformation
    print(f'{qty} {item} cost ${price:.2f}')
ValueError: Unknown format code 'f' for object of type 'str'
```

We received that error due to the word strawberries being attempted to be converted to a `float` for the output screen.

TRY IT

Directions: Try changing the argument location. Do you get the ValueError?

However, if we swapped the `qty` and the `price` in the arguments, we wouldn't get an error.

EXAMPLE

```
def priceInformation(qty, item, price):
    print(f'{qty} {item} cost ${price:.2f}')

priceInformation(5.99, 'strawberries', 12)
```

The output screen looks like this:

```
5.99 strawberries cost $12.00
```

Rather, this becomes a logical error that we would have to identify when testing the code. It's always a good idea to double-check the order of what's being passed into a function to ensure that it is what you expect to see.

TRY IT

Directions: Try changing the qty and price argument location. Are you getting 5.99 (almost 6 full strawberries) for $12 dollars? Wow, they are a little expensive. And 5.99 strawberries? Maybe one had a small bite out of it... You can bypass this issue by identifying the arguments by the parameter name. This allows you to place them out of order, as Python is able to use the variable names provided to match the values with the arguments like this:

EXAMPLE

```
def priceInformation(qty, item, price):
    print(f'{qty} {item} cost ${price:.2f}')

priceInformation(price=5.99, item='strawberries', qty=12)
```

The output screen looks good again.

```
12 strawberries cost $5.99
```

In order to do so, we include the parameter name with the equal sign and the value to be passed in.

TRY IT

Directions: Try changing the code to include the equal signs for the arguments. Now try switching the argument's location. Any issues?

3. Scope Errors

Another common issue when it comes to functions is scope rules. Variables that are defined inside the body of a function have local scope and those defined outside of a function body have global scope. For **local scope**, this means that the variables can only be accessed within the block, function, method, or class that they are declared. With **global scope**, variables can be accessed throughout the entire program. We will get into more detail on scope with classes in Unit 3 but for now, let's just see an example of what this means.

EXAMPLE

```
total = 0;

def sum(arg1,arg2):
   total = arg1 + arg2;
   print ("Local total: ",total)
   return total;

sum(10,20);
print ("Global total : ", total)
```

Let's break down this program before running it.

We can see a variable called `total` being declared and set to 0. The `total` variable is a global variable since it exists outside of any function. Then we define a function called `sum()` that takes two arguments (`arg1` and `arg2`). In the body of the `sum()` function, we have a variable called `total` that is set to the sum of the arguments passed into the function. The function then uses a `print()` function to output the total, and finally

we see the return statement returning that value. Then we call the `sum()` function and pass in two integers (`10` and `20`). Finally, we use a `print()` function to output the global `total`.

We could potentially assume that with the way that the code runs, the `total` variable on line 4 (`total = arg1 + arg2`, in the body of the `sum()` function) is making use of the same variable. However, this is the result of running the code.

```
Local total:  30
Global total :  0
```

That's odd, right? What happened in this case? The problem that we've run into is that the `total` variable that's being used within the `sum()` function is a different variable with a local scope. That local variable can only be accessed inside of the function. So even though we have a `total` variable declared globally, they are in fact two separate variables with the same name.

TRY IT

Directions: Try this code containing a global and local variable.

So, in summary on scope:

Global variables are those that we do not define inside any function and have a global scope, whereas local variables are those that are defined inside a function and their scope is limited to that function only. Local variables are accessible inside the function where they were initialized or used, whereas global variables are accessible throughout the entire program and inside all functions.

TERMS TO KNOW

Local Scope
Local scope means that the variables can only be accessed within the block, function, method, or class that they are initialized or used.

Global Scope
Global scope means variables can be accessed throughout the entire program.

SUMMARY

In this lesson, we learned about some of the **common issues with functions** and how many of these issues also affect any other statement or programming block in Python. For example, functions can call one another in the middle of a function, so to help debug a program, it makes sense to follow the flow of execution instead of trying to read a program from top to bottom. We covered **function argument errors** that can appear when multiple parameters are out of order, and we looked at ways to ensure parameter order accuracy by naming the parameter list as an alternative. Finally, we learned about **scope errors** that can occur if a variable that is declared globally is different from one that is declared locally, even if they have the same name.

Best of luck in your learning!

Source: THIS CONTENT AND SUPPLEMENTAL MATERIAL HAS BEEN ADAPTED FROM “PYTHON FOR EVERYBODY” BY DR. CHARLES R. SEVERANCE ACCESS FOR FREE AT www.py4e.com/html3/ LICENSE: CREATIVE COMMONS ATTRIBUTION 3.0 UNPORTED.

TERMS TO KNOW

Flow of Execution

The order in which statements are executed is called the flow of execution. Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Global Scope

Global scope means variables can be accessed throughout the entire program.

Local Scope

Local scope means that the variables can only be accessed within the block, function, method, or class that they are initialized or used.