8. Functions in Python 3

8.1 Introduction

A function is a block of instructions that performs an action and, once defined, can be reused. Functions make code more modular, allowing you to use the same code over and over again.

Python has a number of built-in functions that you may be familiar with, including:

- `print()` which will print an object to the terminal
- `int()` which will convert a string or number data type to an integer data type
- `len()` which returns the length of an object

Function names include parentheses and may include parameters. In this tutorial, we'll go over how to define your own functions to use in your coding projects.

8.2 Defining a Function

Let's start with turning the classic "Hello, World!" program into a function.

We'll create a new text file in our text editor of choice, and call the program `hello.py`. Then, we'll define the function.

A function is defined by using the `def` keyword, followed by a name of your choosing, followed by a set of parentheses which hold any parameters the function will take (they can be empty), and ending with a colon.

In this case, we'll define a function named `hello()`:

hello.py

```
def hello():
```

This sets up the initial statement for creating a function.

From here, we'll add a second line with a 4-space indent to provide the instructions for what the function does. In this case, we'll be printing `Hello, World!` to the console:

hello.py

```
def hello():
    print("Hello, World!")
```

Our function is now fully defined, but if we run the program at this point, nothing will happen since we didn't call the function. So, outside of our defined function block, let's call the function with `hello()`:

hello.py

```
def hello():
    print("Hello, World!")
hello()
```

Now, let's run the program. You should receive the following output:

```
Output
Hello, World!
```

Functions can be more complicated than the `hello()` function we defined above. For example, we can use `for` loops, conditional statements, and more within our function block.

For example, the function defined below utilizes a conditional statement to check if the input for the `name` variable contains a vowel, then uses a `for` loop to iterate over the letters in the `name` string.

names.py

```
# Define function names()
def names():
    # Set up name variable with input
    name = str(input('Enter your name: '))
    # Check whether name has a vowel
    if set('aeiou') & set(name.lower()):
        print('Your name contains these vowels.')
        print( set('aeiou') & set(name.lower()) )
    else:
        print('Your name does not contain a vowel.')
# Call the function
names()
```

The `names()` function we defined above sets up a conditional statement showing how code can be organized within a function definition.

Defining functions within a program makes our code modular and reusable so that we can call the same functions without rewriting them.

8.3 Working with Parameters

So far we have looked at functions with empty parentheses that do not take arguments, but we can define parameters in function definitions within their parentheses.

A **parameter** is a named entity in a function definition, specifying an argument that the function can accept.

Passing Arguments by Value vs. by Reference

In Python:

- Immutable arguments (such as integers, floats, strings and tuples) are *passed by value*. That is, a copy is cloned and passed into the function, and the original cannot be modified inside the function.
- Mutable arguments (such as lists, dictionaries, sets and instances of classes) are *passed by reference*. That is, they can be modified inside the function.

Let's create a small program that takes in parameters `x`, `y`, and `z`. We'll create a function that adds the parameters together in different configurations. The sums of these will be printed by the function. Then we'll call the function and pass numbers into the function.

add_numbers.py

```
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    print(a, b, c)


add_numbers(1, 2, 3)
```

We passed the number `1` in for the `x` parameter, `2` in for the `y` parameter, and `3` in for the `z` parameter. These values correspond with each parameter in the order they are given.

The program is essentially doing the following math based on the values we passed to the parameters:

```
a = 1 + 2
b = 1 + 3
c = 2 + 3
```

The function also prints `a`, `b`, and `c`, and based on the math above we would expect `a` to be equal to `3`, `b` to be `4`, and `c` to be `5`. Let's run the program

```
Output
3 4 5
```

When we pass `1`, `2`, and `3` as parameters to the `add_numbers()` function, we receive the expected output. Parameters are arguments that are typically defined as variables within function definitions. They can be assigned values when you run the method, passing the arguments into the function.

8.4 Keyword Arguments

In addition to calling parameters in order, you can use **keyword arguments** in a function call, in which the caller identifies the arguments by the parameter name.

When you use keyword arguments, you can use parameters out of order because the Python interpreter will use the keywords provided to match the values to the parameters.

Let's create a function that will show us profile information for a user. We'll pass parameters to it in the form of `username` (intended as a string), and `followers` (intended as an integer).

profile.py

```
# Define function with parameters
def profile_info(username, followers):
    print("Username: " + username)
    print("Followers: " + str(followers))
```

Within the function definition statement, `username` and `followers` are contained in the parentheses of the `profile_info()` function. The block of the function prints out information about the user as strings, making use of the two parameters.

Now, we can call the function and assign parameters to it:

profile.py

```
def profile_info(username, followers):
    print("Username: " + username)
    print("Followers: " + str(followers))

# Call function with parameters assigned as above
profile_info("sammyshark", 945)

# Call function with keyword arguments
profile_info(username="AlexAnglerfish", followers=342)
```

In the first function call, we have filled in the information with a username of `sammyshark` and followers being `945`, in the second function call we used keyword arguments, assigning values to the argument variables.

Output

```
Username: sammyshark
Followers: 945
Username: AlexAnglerfish
Followers: 342
```

The output shows us the usernames and numbers of followers for both users. This also permits us to modify the order of the parameters, as in this example of the same program with a different call:

profile.py

```
def profile_info(username, followers):
    print("Username: " + username)
    print("Followers: " + str(followers))
```

```
# Change order of parameters
profile_info(followers=820, username="cameron-catfish")
```

When we run the program again, we'll receive the following output:

```
Output
Username: cameron-catfish
Followers: 820
```

Because the function definition maintains the same order of `print()` statements, if we use keyword arguments, it does not matter which order we pass them into the function call.

8.5 Default Argument Values

We can also provide default values for one or both of the parameters. Let's create a default value for the `followers` parameter with a value of `1`:

profile.py

```
def profile_info(username, followers=1):
    print("Username: " + username)
    print("Followers: " + str(followers))
```

Now, we can run the function with only the username function assigned, and the number of followers will automatically default to 1. We can also still change the number of followers if we would like.

profile.py

```
def profile_info(username, followers=1):
    print("Username: " + username)
    print("Followers: " + str(followers))

profile_info(username="JOctopus")
profile_info(username="sammyshark", followers=945)
```

When we run the program with the `python profile.py` command, we'll receive the following output:

```
Output
Username: JOctopus
Followers: 1
Username: sammyshark
Followers: 945
```

Providing default parameters with values can let us skip defining values for each argument that already has a default.

8.6 Returning a Value

You can pass a parameter value into a function, and a function can also produce a value. A function can produce a value with the `return` statement, which will exit a function and *optionally* pass an expression back to the caller. If you use a `return` statement with no arguments, the function will return `None`.

So far, we have used the `print()` statement instead of the `return` statement in our functions. Let's create a program that instead of printing will return a variable.

In a new text file called `square.py`, we'll create a program that squares the parameter `x` and returns the variable `y`. We issue a call to print the `result` variable, which is formed by running the `square()` function with `3` passed into it.

square.py

```
def square(x):
    y = x ** 2
    return y

result = square(3)
print(result)
```

We can run the program and see the output:

Output

```
9
```

The integer `9` is returned as output, which is what we would expect by asking Python to find the square of 3. To further understand how the `return` statement works, we can comment out the `return` statement in the program:

square.py

```
def square(x):
    y = x ** 2
    # return y

result = square(3)
print(result)
```

Now, let's run the program again:

Output

```
None
```

Without using the `return` statement here, the program cannot return a value so the value defaults to `None`. As another example, in the `add_numbers.py` program above, we could swap out the `print()` statement for a `return` statement.

add_numbers.py

```python
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    return a, b, c

sums = add_numbers(1, 2, 3)
print(sums)
```

Outside of the function, we set the variable `sums` equal to the result of the function taking in `1`, `2`, and `3`as we did above. Then we called a print of the `sums` variable.

Let's run the program again now that it has the `return` statement:

```
Output
(3, 4, 5)
```

We receive the same numbers `3`, `4`, and `5` as output that we received previously by using the `print()` statement in the function. This time it is delivered as a tuple because the `return` statement's expression list has at least one comma.

Functions exit immediately when they hit a `return` statement, whether or not they're returning a value.

return_loop.py

```python
def loop_five():
    for x in range(0, 25):
        print(x)
        if x == 5:
            # Stop function at x == 5
            return
    print("This line will not execute.")

loop_five()
```

Using the `return` statement within the `for` loop ends the function, so the line that is outside of the loop will not run. If, instead, we had used a `break` statement, only the loop would have exited at that time, and the last `print()` line would run.

The `return` statement exits a function, and may return a value when issued with a parameter.

8.7 Using **`main()`** as a Function

Although in Python you can call the function at the bottom of your program and it will run (as we have done in the examples above), many programming languages (like C++ and Java) require a `main` function in order to execute. Including a `main()` function, though not required, can structure our Python programs in a logical way that puts the most important components of the program into one function. It can also make our programs easier for non-Python programmers to read.

We'll start with adding a `main()` function to the `hello.py` program above. We'll keep our `hello()` function, and then define a `main()` function:

hello.py

```
def hello():
    print("Hello, World!")


def main():
```

Within the `main()` function, let's include a `print()` statement to let us know that we're in the `main()` function. Additionally, let's call the `hello()` function within the `main()` function:

hello.py

```
def hello():
    print("Hello, World!")



def main():
    print("This is the main function")
    hello()
```

Finally, at the bottom of the program we'll call the `main()` function:

hello.py

```
def hello():
    print("Hello, World!")


def main():
    print("This is the main function.")
    hello()


main()
```

At this point, we can run our program:

We'll receive the following output:

```
Output
This is the main function.
Hello, World!
```

Because we called the `hello()` function within `main()` and then only called `main()` to run, the `Hello, World!` text printed only once, after the string that told us we were in the main function.

Next we're going to be working with multiple functions, so it is worth reviewing the variable scope of global and local variables. If you define a variable within a function block, you'll only be able to use that variable within that function. If you would like to use variables across functions it may be better to declare a global variable.

In Python, `'__main__'` is the name of the scope where top-level code will execute. When a program is run from standard input, a script, or from an interactive prompt, its `__name__` is set equal to `'__main__'`.

Because of this, there is a convention to use the following construction:

```
if __name__ == '__main__':
    # Code to run when this is the main program here
```

This lets program files be used either:

- as the main program and run what follows the `if` statement
- as a module and not run what follows the `if` statement.

Any code that is not contained within this statement will be executed upon running. If you're using your program file as a module, the code that is not in this statement will also execute upon its import while running the secondary file.

Let's expand on our `names.py` program above, and create a new file called `more_names.py`. In this program we'll declare a global variable and modify our original `names()` function so that the instructions are in two discrete functions.

The first function, `has_vowel()` will check to see if the `name` string contains a vowel.

The second function `print_letters()` will print each letter of the `name` string.

more_names.py

```
# Declare global variable name for use in all functions
name = str(input('Enter your name: '))


# Define function to check if name contains a vowel
def has_vowel():
    if set('aeiou') & set(name.lower()):
        print('Your name contains a vowel.')
    else:
        print('Your name does not contain a vowel.')

# Iterate over letters in name string
def print_letters():
    for letter in name:
        print(letter)
```

With this set up, let's define the `main()` function which will contain a call to both the `has_vowel()` and the `print_letters()` functions.

more_names.py

```
# Declare global variable name for use in all functions
name = str(input('Enter your name: '))

# Define function to check if name contains a vowel
def has_vowel():
    if set('aeiou') & set(name.lower()):
        print('Your name contains a vowel.')
    else:
        print('Your name does not contain a vowel.')

# Iterate over letters in name string
def print_letters():
    for letter in name:
        print(letter)

# Define main method that calls other functions
def main():
    has_vowel()
    print_letters()
```

Finally, we'll add the `if __name__ == '__main__':` construction at the bottom of the file. For our purposes, since we have put all the functions we would like to do in the `main()` function, we'll call the `main()` function following this `if` statement.

more_names.py

```
# Declare global variable name for use in all functions
name = str(input('Enter your name: '))



# Define function to check if name contains a vowel
def has_vowel():
    if set('aeiou') & set(name.lower()):
        print('Your name contains a vowel.')
    else:
        print('Your name does not contain a vowel.')

# Iterate over letters in name string
def print_letters():
    for letter in name:
        print(letter)



# Define main method that calls other functions
def main():
    has_vowel()
    print_letters()

# Execute main() function
if __name__ == '__main__':
    main()
```

We can now run the program: The program will show the same output as the `names.py` program, but here the code is more organized and can be used in a modular way without modification.

If you did not want to declare a `main()` function, you alternatively could have ended the program like this:

more_names.py

```
if __name__ == '__main__':
    has_vowel()
    print_letters()
```

Using `main()` as a function and the `if __name__ == '__main__':` statement can organize your code in a logical way, making it more readable and modular.

8.3 Function Overloading

Python does NOT support Function Overloading like Java/C++ (where the same function name can have different versions differentiated by their parameters).

The `pass` statement

The pass statement does nothing. It is sometimes needed as the statement placeholder to ensure correct syntax, e.g.,

```
def my_fun():
    pass      # To be defined later, but syntax error if empty
```