HyperionDev

Programming With User-Defined Functions

Task

Visit our website

Introduction

This lesson will focus on teaching you how to create your own functions. Functions are the backbone of code, empowering you to tackle complex tasks with elegance and precision. You will also learn how functions can be used to compute certain values using list elements and text file contents.

Defining a function

A function can be defined as follows:

```
# This function, add_one, takes a single parameter x and returns the value of x incremented by 1

def add_one(x):
    y = x + 1
    return y
```

The function that has been created in the example above is defined by the keyword `def` and is named `add_one`. It takes the parameter `x` as input. A **parameter** is a variable that is declared in a function definition. Parameters store the data needed to perform the logic that the function specifies.

Parameters are filled when data is passed to the function as an argument when the function is called (which you will learn about soon). The code indented under `def add_one` is the logic of the function. It defines what happens when the function is called. You can pretty much do anything you want in a function. For example, you can create new data structures, use conditionals, etc.

The function in the example above computes a new variable, `y`, which is the value stored in variable `x` with `1` added. It will then `return` the value `y`.

The general syntax of a function in Python is as follows:

```
def functionName(parameters):
      statements
      return (expression)
```

The def and return keywords

Note the `def` keyword. Python knows you're defining a function when you start a line with this keyword. After the keyword, `def`, put a function name, its input parameters, and then a colon, with the logic of the function indented underneath.

The value after the `return` statement, resulting from `expression`, will be returned/passed back to whatever code 'called' the function, at which point the execution of that block of code stops. Note that the `return` keyword doesn't have to be included in a function. You can write functions that return (send back) a value to the calling function as in the syntax example above, or you can write functions that just **do** something (such as calculate or print) and don't return anything. Before we go into this any further, let's look at what it means to call a function.

Print versus return

Let's look at the differences between `print` and `return`:

- **`print` outputs data to the console.** It's useful for debugging and displaying information to the user.

```
def greet(name):
    # This function takes a 'name', and prints a personalised greeting.
    print(f"Hello, {name}!")

greet("Alice")  # This will print "Hello, Alice!" to the console.
```

- **`return` sends data back to the caller of the function.** It's used when you need to reuse the result of a function elsewhere in your program.

```
def add(a, b):
    # This function takes two arguments, a and b, and returns their sum.
    return a + b
result = add(3, 4)

print(result)  # Print the value of 'result' (which is 7) to the console.
```

In summary, `print` is for displaying information, while `return` is for passing data from a function to the rest of the program. The returned value is stored in a variable, which can then be used elsewhere in the program **or** printed to the console.

Calling a function

It is good practice to define all your functions at the top of your code file and 'call' them as needed later in the code file. You can call a function by using the function's name, followed by the values you would like to pass to the parameters in parentheses. The values that you pass to the function are referred to as **arguments.**

Here is an example of calling our `add_one` function, and assigning the value returned from it to a variable:

```
def add_one(x):
    # Function to add 1 to x
    y = x + 1
    return y

# Call add_one with 5 and store the results in a variable
result_num = add_one(5)

# Output the results, which is 6
print(result_num)
```

You can define a function, but it will not run unless called somewhere in the code. For example, although we have defined the function `add_one` above, the code indented underneath will never be executed unless another line that calls `add_one` with the command `add_one(some_variable)` is added somewhere in the main body of your code.

Working with function parameters

In the function definition, the **parameters** are between the parentheses after the function name. You can have more than one of these variables or parameters – simply separate them by commas. When you call a function, you place the value you would like to pass to the function as an **argument** in parentheses after the function name, e.g.

```
result_num = add_one(5)
```

(If we didn't do something with the returned value, such as set up a variable to hold it, or a print statement to output it, what do you think would happen?)

Now, let's look at another example showing something very similar to our first function-calling example, but with the code written to be shorter and more efficient (more succinct code can be more challenging for beginners to understand, but it's worth noting when code can be written more efficiently, as this is what programmers ultimately aspire to achieve).

In the example below, the `add_one` function is called again. Here, we pass the value `10` as an **argument** to the function. As we created a parameter called `x` when we defined the function `add_one`, passing the argument `10` to the function will result in the parameter `x` being assigned the value `10`.

```
def add_one(x):
    # Function to add 1 to x
    y = x + 1
    return y

num = 10

# Call the function within the print statement and output the result
print("10 plus 1 is equal to: " + str(add_one(num))+".")

# Alternatively
num_plus_one = add_one(10)

# Use the stored result in the print statement
print("10 plus 1 is equal to: " + str(num_plus_one)+".")
```

Think of a call to the function – e.g. `add_one(num)` – as a placeholder for some computation. The function will run its code and return its result in that place.

The above examples have all shown functions that return a value. Let's look at an example that also prints out the result of adding one to an argument that was passed in to it, but does not return anything.

```
def add_one(x):
    # Function to add 1 to x
    y = x + 1
    # Print the results immediately
    print(y)

# Call the function with 3 as the argument, which prints 4
add_one(3)
```

In the above example, `add_one` does not return anything at all, i.e. if you tried to write `result = add_one(3),` nothing would be assigned to `result`. If you then tried to print out `result`, the output from that print statement would simply say `"None"`. Go ahead, try doing this and running it to see for yourself.

Transitioning from sequential to procedural programming

A major switch happens in how you're able to program when you learn about functions. Before now, all your programs have been sequential. This means that code is always executed in the same order in which we read it; from the top of the file to the bottom.

With functions, we lose this. You can define a function anywhere in your file, but it will not run unless it's called somewhere. This means that the statements in your code are no longer necessarily executed in the same order that they are written. This may sound unnecessarily confusing, but you will get used to it quite quickly, and the many benefits of functions outweigh the slight inconvenience of needing to shift your thinking. Let's consider some of these benefits.

Why use functions?

There are many benefits to using functions:

- Creating functions allows you to have **reusable code**. There are many tasks that, as a programmer, you may need to code repeatedly. For instance, say you wrote several lines of code that, given a filename, can open the file, read its contents, and print out its contents to the screen. It may be useful to 'save' that code somewhere so you can easily reuse it. A programmer can define a function named `read_file`, that would encode this logic. That way, the next time they needed to read the contents of a file, they would simply call the function `read_file`. This will return the result of that function, which in this case will result in the output being printed on the screen.

- Functions also make **error-checking and validating your code easier.** Each module can be tested separately, possibly by different developers.

- Functions **divide your code into manageable chunks** to make the code easier to understand and troubleshoot.

- Modular programming is where a different developer or team of developers can code each module (set of functions). Modular programming enables **more rapid application development.** This means that many modules can be developed simultaneously, increasing the speed at which teams can develop applications. Also, developers can reuse existing modules in new applications, leading to more rapid software development.

- Using functions can also make it **easier to maintain applications.** If a part of a system needs to be updated, the whole program doesn't need to be modified. Instead, just the necessary function or functions can be changed.

Scope

Scope is a program's ability to find and use variables in a program. The rule of thumb is that a function is covered in one-way glass: it can see out, but no one can see in. This means that a function can call variables that are outside the function, but the rest of the code cannot call variables that are defined inside the function.

Let's look at an example:

```
# Accesses the internal variable 'total' and also the external variable
'description'
def adding(a, b):
    total = a + b
    return (description + str(total))
x = 2
y = 3
description = "Total: "

# Calls the adding function with x and y as arguments
sum = adding(x, y)
# Prints the results
print(sum)
```

Output:

```
Total: 5
```

In the code example above, the function makes use of the `description` variable from inside the function, despite the fact that this variable is not part of the function, but is outside it. This shows that the function can look outside and use variables from outside the function. Now let's see what happens if we put `description` inside the function:

```
# Internal variable 'description', not accessible outside the function
def adding(a, b):
    total = a + b
    description = "Total: "
    return (str(total))

x = 2
y = 3

# Call adding function with x and y
sum = adding(x, y)

# Attempt to print the results(this will cause an error as 'description is
not defined outside the function'
print(description + sum)
```

Output:

```
NameError: name 'description' is not defined
```

See how the program complains that it can't find the `description` variable? That's because of the 'one-way glass': the rest of the code can't see into the function and so doesn't know that a `description` variable exists.

Default values

When creating your own functions, it is possible to create default arguments. Let's look at the example below:

```
# Function to multiply two numbers, default value of 5 for num2
def multiply(num1,num2=5):
    total = num1 * num2
    print(f"{num1} * {num2} = {total}")

# Uses default num2, Output: 6 * 5 = 30
times_5 = multiply(6)
```

Here we have a function that multiplies two numbers and prints a string with the total. The default value of `num2` is five, so when we call the function and give the argument six, that means that `num1 = 6` – we don't need to give an argument for `num2`; it will default to five. Therefore, when we print out `times_5`, the output will be:

```
6 * 5 = 30
```

What would happen if we took the default value for num2 out and called the function with only one parameter? Let's have a look:

```
# Sets a default value of five for num2
def multiply(num1,num2):
    total = num1 * num2
    print(f"{num1} * {num2} = {total}")

times_5 = multiply(6)
```

Output:

```
TypeError: multiply() missing 1 required positional argument: 'num2'
```

As you can see, an error will be generated explaining that one input argument, in this case `num2`, is missing and the function cannot run without it.

What if you want a function to have a default value, but you sometimes want to override that default value and use a different value as `num2`? It is possible to change the

value of `num2`. At the time you call the function, simply by providing a second argument value. Have a look at the example below:

```
def multiply(num1,num2 = 5):
    total = num1 * num2
    print(f"{num1} * {num2} = {total}")

# Overrides fault; output: 6 * 7 = 42
times_7 = multiply(6, 7)
```

Here, even though `num2` still has a default value of five, we have overwritten that to give it a value of seven.

Now, the output will be:

```
6 * 7 = 42
```

We could also call the function using keyword arguments, so the order in which we write the arguments doesn't matter. For instance, using the above function:

```
times_9 = multiply(num2=6, num1=9)
```

Output:

```
9 * 6 = 54
```

Let's look at a summary of all the variations in how we can call a function with default parameters:

```
# Both parameters have default arguments
def multiply(num1 = 6, num2 = 5):
    total = num1 * num2
    print(f"{num1} * {num2} = {total}")

# If you provide no arguments both defaults are used
multiply_test = multiply()
# If you provide one argument it goes into the first variable and overwrites the
six
multiply_test = multiply(1)
# If two arguments are provided both defaults are overwritten
multiply_test = multiply(2,7)
# If you specify the parameter names, this enables you to provide values in a
different order
multiply_test = multiply(num2 = 8, num1 = 7)
```

```
#  If you did this and didn't specify the name of the other parameter, an error would be generated
multiply_test = multiply(num2 = 8, 7)
```

Output:

```
6 * 5 = 30
1 * 5 = 5
2 * 7 = 14
7 * 8 = 56
SyntaxError: positional argument follows keyword argument
```

Now that you know how to write functions let's explore how you document what the function does. This may seem obvious for these simple examples but as your functions become more complex you need to ensure someone else (including your future self) can easily understand the purpose of a function.

Docstrings

Docstrings are special comments that provide documentation for functions, classes, and modules. They are enclosed in triple quotes (""") and appear as the first statement within the definition. Docstrings serve as a valuable resource for understanding the purpose, parameters, and return values of code elements. They are essential for maintaining code readability and making it easier for others to use and understand your code.

```
def square(length):
    """
    Calculate the area of a square.

    Parameters:
    length (float): The length of one side of the square.

    Returns:
    float: The area of the square.
    """

    return length * length
```

Instructions

Read and run the accompanying **example files** provided before doing the task in order to become more comfortable with the concepts covered in this task.

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task 1

1. Create a Python file called **holiday.py.**
2. Your task will be to calculate a user's total holiday cost, which includes the plane cost, hotel cost, and car rental cost.
3. First, get the following user inputs:
    - `city_flight`: The city they will be flying to (you can create some options for them. Remember, each city will have different flight costs).
    - `num_nights`: The number of nights they will be staying at a hotel.
    - `rental_days`: The number of days for which they will be hiring a car.
4. Next, create the following four functions:
    - `hotel_cost()`: This function will take `num_nights` as an argument and return a total cost for the hotel stay (you can choose the price per night charged at the hotel).
    - `plane_cost()`: This function will take `city_flight` as an argument and return a cost for the flight. Hint: use **if/else** statements in the function to retrieve a price based on the chosen city.
    - `car_rental()`: This function will take `rental_days` as an argument and return the total cost of the car rental (you can choose the daily rental cost).
    - `holiday_cost()`: This function takes three arguments: `num_nights`, `city_flight`, and `rental_days`. Using these three arguments, call the `hotel_cost()`, `plane_cost()`, and `car_rental()` functions with their respective arguments, and finally return the total cost for the holiday.
5. Print out all the details about the holiday in a way that is easy to read.

Try running your program with different combinations of input to show its compatibility with different options.

Be sure to place files for submission inside your **task folder** and click "**Request review**" on your dashboard.

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Do you think we've done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.