Writing Functions in Python

Functions

In computer programming, a *function* is a collection of code which can be applied and reapplied to complete repetitive or similar tasks. In general, functions can be thought of as a kind of recipe where you plug in a set of ingredients, or parameters, and the function follows the recipe so that you don't have to rewrite the same code every time you need to perform the same operation.

Functions make the process of working with code easier for both programmers and readers. For programmers, using functions can replace the need to copy and paste blocks of code and can reduce several lines of code to one or two lines. For readers, functions can make the process of working with someone else's code much simpler, since function definitions are distinct from the areas of the code that are more specific. The separation between a function's definition and the rest of the code can make debugging and editing code easier. Using functions can also make your code run faster, easing the burden on both the Python interpreter and your computer.

Overall, functions are a great way to streamline your code and to make it run more efficiently. Their ability to tackle repetitive tasks makes them a central part of programming.

Python Function Structure

In Python, functions follow a set structure which signals to the Python interpreter that you are designating a block of code as a function. Functions are also given a name, which makes the process of using the function relatively straightforward, since you use the function name much like you would a variable.

Defining a Function

In order to write a function, you begin by writing the header. All function headers begin with the `def` call (which signals to the interpreter that you are about to define a function), name the function, list any parameters in parentheses after the name, and end with a colon. You then write the body, which defines what the function will do. A function definition outlines exactly what operations you want the function to perform, what kinds of variables the function will accept as input, and what you will name the function:

Developed by: Avery Blankenship and Sara Morrell
Questions? Contact us! nulab.info@gmail.com

```
def our_function(parameter1, parameter2):
     some_data = parameter1 + parameter2
     return some_data
```

Function Placement in Code

Like all other Python code, indentation matters in function definitions. The header, where you define the function's name as well as the function's parameters, should be unindented and all subsequent lines that are part of the function body should be indented directly below. Additionally, Python is a programming language that processes code from top to bottom, meaning that function definitions must occur *before* you use the function in your code. Often, programmers place function definitions at the top of their code, after their library import statements, and place all subsequent code below the set of function definitions like below:

```
import library                          ← import statements
import another_library

def function1(parameter):               ← function definitions
     result_one = parameter + 2
     return result_one

def function2(parameter):
     result_two = parameter + 3
     return result_two

x = 1                                   ← the rest of the code
y = 2
function1(x)
Output: 3
function2(y)
Output: 5
```

Parameters vs. Arguments

In addition to giving the function a name—which you will be able to call much like a variable after you have defined the function—the header is where you define the *parameters* of the function. Parameters are placeholder variables which stand in for the data you will pass to a function. When the function is actually called in the code, the data you pass to the function are called *arguments*. The best way to understand the difference between the two is that parameters are in the definition and arguments are the actual data being given to the function.

```
def function1(parameter):    ← this is a parameter, a stand-in for an argument
     result = parameter + 1
     return result

function1(3)    ← this is an argument, the data you want the function to process
Output: 4
```

Return Statements and Local Variables

After you write the header, you write the body of the function as you normally would write code, but you use the parameter names in place of where arguments would be passed into the code. Remember that function definitions are *definitional,* which means that they are primarily describing how you want the interpreter to handle data. They don't work with actual data or even run until you actually call the function and pass it arguments. Similarly, any variables you define within a function cannot be accessed outside of a function unless you use the `return` call to intentionally give the rest of the code access to a variable. Variables used within a function are called *local* variables, meaning that they are only known, locally, to the function where they are defined.

```
def function1(parameter):
     x = 1                     ← this line will reset x to 1 every time the function is called
     x = x + parameter
     print(x)                  ← this line will print x plus the parameter value

print(x + 1)                   ← this line would give you an error: x is only defined
                                 within the function
Output: NameError: name 'x' is not defined
```

Typically, `return` is used at the end of a function definition to make the results of the function accessible after the function is done running. When you use `return`, anything that you `return` will be accessible to the rest of the code, but as raw data, not as a stored variable. If a function is like a recipe, the return call allows the food to leave the kitchen, but it doesn't put the food on a plate—like carrying soup in your bare hands. Storing the results of a function in a new variable outside of the function is like pouring the soup into a bowl before trying to serve it. For example:

```
bowl = 'empty'
ingredient = 'carrot'
def recipe(ingredient):
     cooking = 'vegetable_soup'
     soup = ingredient + cooking
     return soup
print(bowl)   ← since you haven't saved the function results in a variable, the bowl
              is still empty
Output: empty
print(recipe(ingredient))← will print the function results, but they aren't saved
                         in a variable
Output: carrotvegetable_soup
bowl = recipe(ingredient)
print(bowl)             ← since the function results are saved in the variable
                        bowl, the bowl is full of soup
Output: carrotvegetable_soup
```

Best Practices for Writing a Function

The best place to start when writing a function is to outline what it is you want the function to actually *do*. Often, this outlining process involves taking a somewhat specific task and generalizing it enough that it could reasonably be transformed into a function and thus become applicable to other kinds of tasks. For example, we might want to make our recipe function general enough that we can add ingredients other than carrots to our soup.

Translating the function outline into pseudocode is a good next step before you write out the code, as it allows you to work through how the code will move through the function before you worry about the actual syntax of Python. Pseudocode is a term that refers to structured notes that resemble code, but aren't actual executable code—these notes often resemble code, but aren't restricted to the actual vocabulary of Python and thus the interpreter wouldn't understand them. Pseudocode often gives you the space to work through complicated problems and to ensure *you* fully understand a problem and how you are designing its solution before giving your solution to the computer. Below is an example of pseudocode for the above recipe function:

1. Define the ingredient to add to the soup → ingredient = some vegetable
2. Define the recipe function → def function_name(ingredient):
    a. Define what you are cooking → cooking = name of dish

   b. Combine the ingredient with what you are cooking and save soup after done cooking → soup = ingredient + cooking
3. Return the soup → return soup

Finally, testing out different parts of the function in chunks of code before wrapping all of the code in a function definition can allow you to make sure that the individual pieces of the code are working properly before you try to run it all together. Remember that functions only execute when you call them in code, which can make the debugging process difficult if you don't test the function code in pieces prior to wrapping it all up in a definition statement. Sprinkling print statements throughout the function can also help you figure out where the code isn't working by paying attention to which of the statements actually print and which statements the function skips over or gets stuck on. Ultimately, this drafting process not only sets you up to better debug your own code, but can also train you to understand the code of others much faster.

Learn More

"Python functions." *Microsoft*, https://learn.microsoft.com/en-us/training/modules/functions-python/. Accessed 11 June 2024.

"Python Functions." *W3 Schools*, https://www.w3schools.com/python/python_functions.asp. Accessed 11 June 2024.

Turkel, William and Adam Crymble. "Code Reuse and Modularity in Python." *Programming Historian*, 17 July 2012, https://programminghistorian.org/en/lessons/code-reuse-and-modularity. Accessed 11 June 2024.

Developed by: Avery Blankenship and Sara Morrell
Questions? Contact us! nulab.info@gmail.com