Bramble Eagle-Jack

2021-02-15

Foundations Of Programming - Python

Assignment 6 Knowledge Document

Functions

# Introduction

This week we’re asked to review learning materials that give us an in-depth look at functions. The assignment asks us to convert blocks of code in the program body of a starter script to functions while paying special attention to separation of concerns.

# Functions

Functions are named blocks of code that are programmer-defined. The def keyword is used to tell Python that the following block of code will be defined as a developer named function. Function naming should follow variable naming conventions and convey what the function does. I can repeat function calls throughout my code in place of the grouped statements contained within making for cleaner main program code. Also, if I need to make changes, I’m only changing the function once instead of multiple groups of code throughout the script. I need to make sure that all my functions are defined before I call them and Python does not allow me to create two functions with the same name.

1. **def** hello(name='Claire'):
2. **print**('Hello', name + '!')
3. **return** name, name
5. names = hello('Bramble')
6. **print**('Goodbye', names[0], names[1] + '!')

Figure 1 - Script demonstrating function definition, default argument values, returning function values – formatted at [Planet B](http://www.planetb.ca/syntax-highlight-word)[[1]](#footnote-1)

I can create a docstring with triple quotes inside a function to describe it but it must be done in the first line of the definition. Docstring style is often dictated by the team you’re working with but can include explanation of the function, arguments and return values. Describing the function in a docstring further cleans up a script because, without the function, I would have a chain of comments explaining what a block of code does in the middle of a program.

Like native Python functions, developer-named functions accept arguments, also known as parameters. There is no limit to the number of arguments that a function can accept. Arguments can be made implicitly and rely on positional values or I can make them explicitly and assign values to the parameters when the function is called. I can also set default values when defining the function that are used when the function is called unless an argument is made that overrides the default value.

If I want to use a value created inside a function, I have to return it. A function can return one or more values, but it returns None if there’s no return statement. If I return multiple values, they need to be in a collection separated by commas in the return statement. I need to unpack them like I would a tuple if I want to use them separately in the main body or I can assign multiple values to a single variable and index them to call positional values.

Variables created within a function only have the scope of that function. Even if a variable within a function has the same name as a variable in the main body of a script, called shadowing, the two variables will not affect each other. Only by using the global keyword on a variable within a function can I change a variable that has been defined in the main body. To avoid confusion, it’s best practice to not shadow variables.

1. x = 6
2. y = 2
4. **def** double\_xy():
5. **global** y
6. x = 4
7. x \*= 2
8. y \*= 2
9. **print**(x)
10. **print**(y)
12. double\_xy()
13. **print**(x)
14. **print**(y)

Figure 2 - Script demonstrating variable scope in functions - formatted at [Planet B](http://www.planetb.ca/syntax-highlight-word)[[2]](#footnote-2)

# Assignment Six

I started the assignment by transferring the code block for writing data to file into the FileProcessor class. I named the function write\_file so it would be similar to the already named read\_file and created it to require two arguments, the name of the file being written and the table to be written to file. I then replaced the variables strFileName and lstTbl from the original code block with the arguments required by the function definition, file\_name and table, because the variables would later be used when the function is called as parameters. The function’s purpose is to write data to file so there are no values to return.

Graphical user interface, text

Description automatically generated

Figure 3 - CDInventory.py running in Spyder

Next, I focused on the functions for user input to create new data and adding that data to a 2d table. The former I named input\_inventory and defined it within the IO class because it will be asking the user for input. Not much changed for this function from the original code block; it doesn’t require arguments, but I do have to return the values created from the user inputs so that they can be used in the function for creating the table data. In the main body, I assign the collection of returned values to the variable cdValues in a single tuple and use the indexed values as arguments in the function add\_inventory.

Add\_inventory is a function that creates a dictionary with the user input values and appends it to a list so it is defined in the DataProcessor class. The three values from user input and the list make up the four arguments required for the function when called. Because the original code block uses the integer function on the cd\_id value, I decided to add error handling in the case that the user enters a string for the ID and crashes the program. The new dictionary from user input will only append to the list if the ID is an integer, otherwise the program will tell the user it needs to be and go back to the main menu loop. There are no return values for this function.

Text

Description automatically generated

Figure 4 - CDInventory.py running in terminal

Finally, the function for deleting a CD inventory dictionary from the list is named del\_id and added to the DataProcessor class because it directly affects the table saved to temporary memory. This function requires two arguments, one gathered from the user on input and the other references the list of dictionaries used throughout runtime. The function checks each dictionary’s ‘ID’ key in the list for an integer value that matches user input by iterating through the list with a counter. If the values are equal, then the entire dictionary in the list is deleted based on the counter position, otherwise the user is told that the ID could not be found.

# Summary

This week, the introduction of functions presented a new challenge, different from nested loops or multi-dimensional data. The abstraction that results from creating a main program full of functions makes the code look clear and concise, but it can also be overwhelming, like I moved to a different house and all my boxes are labeled by room, but I don’t know what’s in them. I think I need to seek out more scripts full of functions to read in order to better familiarize myself as we move into the second half of the course.

I think the assignment would have been much more difficult without the TODO markers and I expect there might be an assignment in the future where I can’t rely on them to help. Had they not been there, I think I would have spent a considerable amount of time trying to work the process for loading inventory into a function before realizing that it was already several functions and a few IO not worthy of its own function. When converting code blocks into functions, it can be a struggle to figure out which variables in the block should become arguments that will later be passed when the function is called in the main body. There are also more details than when all I had to worry about was making sure a variable was descriptive; my class, function, arguments need to align with their objective and the docstring goes one-step further than an in-line comment.

Despite the difficulties, seeing a completed program with defined classes and functions that follows the separation of concerns was really encouraging and I’m impressed that the process reduced the code in the main program by almost 30 lines, into something readable.

1. Captured 2021-Feb-21 [↑](#footnote-ref-1)
2. Captured 2021-Feb-21 [↑](#footnote-ref-2)