Name: Matthew Penny

Date: 2021 AUG 13

Course: Foundations Of Programming: Python

Assignment: Assignment 06

**Functions, Classes, and Docstrings**

**Introduction**

In this module, we covered functions, classes, docstrings, and variable scopes. I covered various study materials including videos and web pages and then worked through three labs before a final project. Overall, this module was fairly comprehensive and also clearly demonstrated how functions and classes can clean up your code, make it easier to manage, and follow concepts such as separations of concerns as well.

**Lessons Learned and Concepts of Interest**

As a requirement for this module, I reviewed all videos and content from the module 05 section here: https://saravji.github.io/saravjis\_hut/FDN\_Prog/Modules.html (external site)

I also read chapter five of “Python Programming, Third Edition – For the Absolute Beginner”, reviewed one youtube video that detailed examples of content we were reviewing. After covering these materials, I am able to address points of interest laid out in the course materials:

**What is a function?**

A function is a block of re-usable code that accomplishes a dedicated task or purpose. We covered many examples in the course, but some example functions may be ones such as these:

* A function that takes three pieces of data from a user and combines/formats them into a date string. For example, I give the function the values: “03”, “August”, “2021” and the function returns a string like “03/AUG/2021” as the user needs it.
* A function that splits up strings by a certain character and returns it in a list. For example: “JUL-AUG-SEP” is supplied to a function and it returns it as [‘JUL’,’AUG’,’SEP’]
* There’s also plenty of functions built-in we already have (in addition to custom ones we write) such as .trim(), .split(), .format(), .items() and more.

There are many options and functions give you a way of having many reusable pieces of code at your disposal that accomplish a needed function. This also prevents you from having to continually “reinvent the wheel” or keep redundant blocks of code in your script.

**What are parameters?**

Parameters are representations of arguments or values that are passed to a function. For example, when I declare the function def testFunction(myList, myInt) – the two items in the brackets are “positional variables” that are processed in relation to the order they come in. So in this case, it’s expecting two items: first, the myList object and then the myInt object.

**What are arguments?**

These are the ACTUAL values passed to the function, in the expected order. Using the example above, I might call the function as so and pass it two arguments:

usrLst = [“Freddy”,”Michael”,”Jason”]

usrInt = 6

testFunction(usrLst, usrInt)

In this case, the values within those two variables are the contents of the arguments and will be manipulated within the function as needed.

**What is the difference between parameters and arguments?**

As noted above, parameters are contained within the function definition brackets. By default, they are positional parameters and correspond to actual arguments or values passed in when the function is called or used. This also helps define how the function will work. For example, if we pass in one argument to a function that has two parameters, it will not work (unless a default value has been set for a parameter).

**What are return values?**

As the name implies, these are values returned by a function to the main program that called the function. These returned values can be basically any data type; though a return statement is purely optional which is sometimes the case with parameters as well. For example, I may design a very simple function that takes three parameters and returns them as a formatted string:

name1 = “Freddy”

name2 = “Jason”

name3 = “Michael”

myNames = formatFunction(name1, name2, name3)

names = name1 + “ and “ + name2 + “ and “ + name3

return names

in which case, myNames will now hold the returned value of “Freddy and Jason and Michael”.

**What is the difference between a global and a local variable?**

In terms of scopes, a global variable is essentially one that exists throughout an entire program. If we create variables and don’t have any features such as classes or functions, than the variables are all essentially global. This changes when we discuss functions and classes. Within functions or a class, variables that are declared or created are, by default, local. This means they only exist within the function itself in which they are declared and are not accessible from outside of it. These variables are also typically destroyed once a function completes a task. This can be overridden using the keyword “global” before declaring a variable of the same name in a function, but isn’t recommended. I’ll explain this a bit more in my description of shadowing below.

**What is shadowing?**

Shadowing is the act of naming a variable within a function the same thing as a variable that exists outside of it. For example:

myInt = 530

def myFunction()

myint = 200

Will result in two variables with the same name. The first, a global variable with the int value of 530. The second, a local variable with the int value of 200. After the function runs, the variable declared within the function will be destroyed and the global value declared outside of the function will still exist and hold the same value it always did, regardless of what we did with the variable of the same name inside of the function. Shadowing is also not recommended, as it can make your code more difficult to follow and confusing.

**How do you use functions to organize your code?**

Functions can be used to move your code into dedicated, reusable blocks that accomplish specific tasks. Instead of writing out a function in the main body of your code and then copying or pasting it around, you can put it inside of a function. This will keep your main body of code cleaner – and as noted below, you can take functions that do similar things and put them in a class. Functions cut down on clutter, streamline your code, and overall make it look tidier since a lot of the action is happening “off screen” inside of a function somewhere else.

**What is the difference between a function and a class?**

A class is a feature used to group together like-functions that do similar things. The way I like to think about it is like a toolbox and tools. The toolbox (a class) holds a series of tools (functions) that do a specific thing. For example, a hammer’s function is to drive nails. A screwdriver’s function is to screw screws, and so on. You open your toolbox (class) and select your tool (function) for the needed purpose. In terms of invocation, a function is called from a class by specifying the class name followed by a dot and then the function name: myclass.myfunction()

**How do functions help you program using the “Separations of Concerns" pattern?**

Functions act as another way to put code into compartments – either just as functions or put within a bigger class. Code can be created with a dedicated purpose and then divided into functions that meet separations of concerns. For example, we can create several methods that focus on presenting data to a user and put those in a presentation class. These functions can be called during a dedicated presentation section within the main body of code. This concept can be applied to any sort of category and can help in not only keeping code more readable, but staying within needed or designated areas of the main code body that serve a specific purpose.

**Labs and Examples Covered**

As noted prior, we were given several labs to cover from module 06, here: https://saravji.github.io/saravjis\_hut/FDN\_Prog/Modules.html (external site)

I will discuss in detail these labs below and show output from them.

First, we had lab A. Final code shown below:

A picture containing text, screenshot, monitor, screen

Description automatically generated

***Fig 01. Completed lab A with functions containing a return statement***

For this lab, we were required to change the code to provide a return value and take two parameters. The example code provided did neither and simply manipulated two variables directly before printing them to the terminal. To address this, I had each function take two parameters (numA, numb), do the calculation and assign it to a variable, and then passed it back with a return statement. I then called the functions below and passed in two arguments that are entered in the terminal and had the returned value assigned to four variables that are then printed to the screen. Overall, this lab was pretty straightforward and was a straightforward way of showing how a function works either manipulating values directly, or taking parameters and returning values – which makes them more versatile.

Next, I completed lab B – seen below:

A picture containing text, monitor, screenshot, screen

Description automatically generated

***Fig 02. All the functions from lab A, condensed into a single function***

This lab tasked us with reworking Lab A to have the four separate functions condensed into one. Overall, it was pretty simple and straightforward. I created a single function called “getCalcs” and had the same two parameters passed in. Within, I create four variables based off the four required calculations and then pass all of them back as a tuple. Since a tuple can’t be displayed as needed in the print statements below it, I unloaded the tuple on line 31 to my needed variables and then had them printed. This lab showed us the ability of functions to return multiple variables, as well as how that can possibly complicate things as you end up with a tuple versus a variable with a single value.

Once this lab was completed, I moved on to lab C – seen below:

Text

Description automatically generated

**Fig 03. The first half of the Lab C script, contains two functions in the SimpleMath class with docstrings**

**A picture containing text, screenshot, monitor, screen

Description automatically generated**

**Fig 04. The second half of the Lab C script, contains one function in the SimpleMath class with docstrings and main code**

This lab tasked us with creating four separate functions (from the completed lab A example) and had us enter it into a class. It also dictated that I add docstrings for each function as well. I did this by creating the SimpleMath() class to hold my four functions. These are declared using the @staticmethod identifier before the “def” line. After the def line dictates the function name and parameters, I created a docstring that contains a brief description of what the function does, the arguments it takes, and the value it returns. I then rewrote the lines to call the functions in the classname.function() format and passed two float values into them. The functions perform the calculations and return the necessary values that are then assigned to four variables and displayed to the user.

Overall, this was a fun introduction to classes and docstrings. Recycling some of the old code from previous labs helped me build on the concepts easier and made the transition of using a class simpler to grasp.

Lastly, we had our actual assignment for the module. In this case, we were given a set of existing code. It already had some classes and functions written, but had several “TODO” statements that noted some existing code still had to be moved into functions.

Text

Description automatically generated

**Fig 05. Two added functions in the DataProcessor class that are the result of completed “todo”s.**

Initially, the task seemed a bit overwhelming as the file itself was long and some porting to functions had already been completed. I decided to take it one step at a time and tackled each step by looking at each “todo” item. Initially, I started in the “A” function block and moved the code into two new functions:

Text

Description automatically generated

**Fig 06. The rewritten “A” block where I moved code into two new functions.**

I started by creating the add\_CD() function in the I0 class as a way to gather user input and simply return the values as a tuple that is assigned to usrInput. This is then passed to the new function I made in the DataProcessor class - .add\_dictionary, which takes the tuple and the list of lists as arguments. Within, the tuple is mapped to a dictionary row object that is appended to the list of lists (lstTbl) that was also passed in.

Next, I had to move the “D” option to a function:

Text

Description automatically generated

***Fig 07. The updated “D” section of the code now using the DataProcessor.Delete\_CD() function.***

After the first changes I did in the “A” block, it became easier for me to complete this porting. The Delete\_CD() function simply takes the intIDDel integer for the ID number to be deleted, and the lstTbl. I then simply copied all the existing code into the method. The only thing I had to pay close attention to in this case was what I was passing in as parameters.

Lastly, we were tasked with rewriting the “S” option, shown below

Text

Description automatically generated

***Fig 07. The updated “S” section of the code now using the FileProcessor.write\_file() function.***

This port to a function/class was also straightforward. We just make sure that strFilename (a variable containing the filename to be written to/saved as) and the lstTbl (2d data object) are passed. Once this is done, I simply moved all the existing code to the method itself and had it successfully write out to a file! Runtime screenshots will be included in the appendix.

Lastly, this script is available on github here:

**Summary**

This module was a great introduction to classes and functions and I feel was quite needed with how long and sprawling code is becoming in assignments. I felt the demonstrations and labs were comprehensive with the final assignment being fairly challenging in having to analyze partly completed code. But, that did seem to enforce the lessons taught better. Other concepts such as docstrings and variable scope were great complimentary components to discuss and likely prevented me from making some errors. I look forward to future modules where we reinforce these concepts and build on them.

**Appendix**

**A screenshot of a computer

Description automatically generated with medium confidence**

***Fig 08. Running “I” option to show loaded in inventory***

**Text

Description automatically generated**

**Fig 09. Running “a” option and adding a CD**

**Text

Description automatically generated**

**Fig 10. Running the “d” option and showing results**

**Text

Description automatically generated**

**Fig 11. Running “s” option to save to file.**

**Text

Description automatically generated**

**Fig 12. Running “I” and “a” options in console**

**Text

Description automatically generated**

**Fig 13. Running “d” option in console and showing output**

**Text

Description automatically generated**

**Fig 14. Running “s” option and writing to file, updated txt file on the right.**