Nabil Knight

2022-Mar-06

IT FDN Programming: Python

Assignment 06

Working with Functions, Classes and Docstrings

# Introduction:

During week six we began learning about functions in a lot more depth than the previous week. We covered a number of topics on how to use them within our programs. Some topics included: using positional and named arguments, return values, the ‘None’ keyword, and using reference data types to name a few of the items. Additionally, we were introduced to Classes and how you can use a set of functions within a class to help format your code into logical blocks. And lastly, we learned about docstrings and variable scope. To put our learnings into practice we refactored sample code from last week’s assignment to utilize functions and classes.

# Functions:

As stated in week five, a function is a group of statements defined by a function name. Functions don’t process results until they are called, so they need to be defined prior to calling them. Functions allow you pass in arguments; arguments are values that can be processed within the function. Example below:

**def** **Hitchhiker**(message):

print('search entry for: {}'.format(message))

As you can see above, we are defining the function Hitchhiker which has the argument message. In the print statement we are calling message which we can use to pass another value in to. A programmer can use as many arguments that is needed but it is a best practice to keep this within reason as it can make your code confusing at some point. There are many different ways you can pass in arguments but it is best to follow a standard that your ecosystem of programmers whether it be within your company, school, etc. routinely adheres to. It will allow others to understand your code easier, make for more efficient code reviews, and to stick to a syntax that everyone can agree upon.

Within a function you can use a return value which can be used instantly or assigned to a variable. Capturing the result into a variable allows a programmer to utilize this result more than once. In Lab A we used the simple math program from a few weeks ago to use functions for each of the arithmetic solutions seen below:

#### HIGHLIGHTED CODE

-------- DATA --------#

intNum1 = **None**

intNum2 = **None**

# -------- PROCESSING --------#

#process the data

**def** **getSum**(val1, val2):

result = val1 + val2

**return** result

**def** **getDif**(val1, val2):

result = val1 - val2

**return** result

**def** **getPro**(val1, val2):

result = val1 \* val2

**return** result

**def** **getQuo**(val1, val2):

result = val1 / val2

**return** result

# -------- PRESENTATION (I / O) --------#

#get user input data

print('Basic Math script. Calculating the Sum, Difference, Product and Quotient of two numbers')

intNum1 = int(input('Please enter the first number: '))

intNum2 = int(input('Please enter the second number: '))

#output the results to the user

print('This script is using numbers', intNum1, 'and', intNum2)

print('Sum:\t\t', getSum(intNum1, intNum2))

print('Diff:\t', getDif(intNum1, intNum2))

print('Prod:\t', getPro(intNum1, intNum2))

print('Quot:\t', getQuo(intNum1, intNum2))

In each function we are placing arguments val1 and val2 as variables for the numbers the user enters. In the presentation we call each function with the entered numbers in both arguments.

The ‘None’ keyword is used to define a null value and in python, functions can have none, one or multiple values passed as a result. The ‘None’ keyword is generally used when a return statement has no attributes or there are no return statements within the function. When you return multiple values, you need to aggregate them into a collection and then return that collection.

.format(answer\_one, answer\_two)

Python implicitly packs and unpack tuples. In other languages you have to do this within the code. Python also allows a programmer to use positional and named arguments and can even mix the two methods. But in general, not naming arguments will assign them to their positions, while you need to explicitly name them to have this clarity. You can even use default values for arguments.

**def** **HitchhikerTwo**(value1, value2):

search = 'please search {}'.format(value1)

result = 'result: {}'.format(value2)

answer1, answer2 = HitchhikerTwo(strAnswer, strData)

In Lab B we used our code from Lab A to unpack a tuple and print the results as below:

# -------- DATA --------#

intNum1 = **None**

intNum2 = **None**

getSum = **None**

getDif = **None**

getPro = **None**

getQuo = **None**

# -------- PROCESSING --------#

#process the data

**def** **getArith**(val1, val2):

getSum = val1 + val2

getDif = val1 - val2

getPro = val1 \* val2

**try**:

getQuo = val1 / val2

**except**:

getQuo = 'Error you entered a 0 in the denominator'

**return** getSum, getDif, getPro, getQuo

# -------- PRESENTATION (I / O) --------#

#get user input data

print('Basic Math script. Calculating the Sum, Difference, Product and Quotient of two numbers')

intNum1 = int(input('Please enter the first number: '))

intNum2 = int(input('Please enter the second number: '))

#output the results to the user

print('This script is using numbers', intNum1, 'and', intNum2)

print()

getSum, getDif, getPro, getQuo = getArith(intNum1, intNum2)

print('Sum:\t\t{}\n'

'Diff:\t{}\n'

'Prod:\t{}\n'

'Quot:\t{}\n'

.format(getSum, getDif, getPro, getQuo))

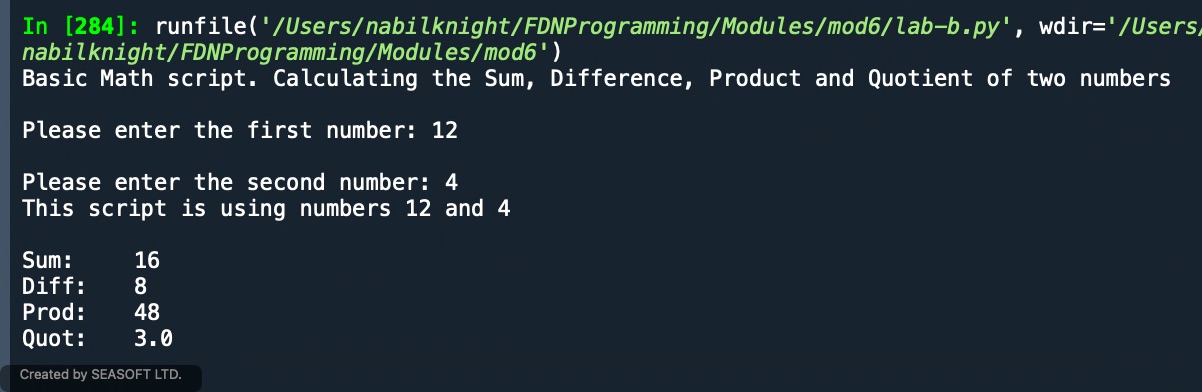
In the above code we defined one function with multiple return results for each of the different math equations. We then called them with one print statement by using a similar method as the HitchHikerTwo example. Python unpacks the tuple automatically as seen below:

Figure 1 - Lab B Results

As stated earlier, Python allows for default values within an argument. This means that if no value is passed from the result, it will pass the default set. And you can also use the None keyword within a function (not just set as a variable) to change what happens within the function. So, if no value is passed in, the None keyword could be used as a check to ensure the data is passed through to allow the function to work properly.

As discussed in week two, Python uses value and reference data types. Simple data ie. x=3 is known as a value data type and complex data x = [1, 2] is known as reference data. Basically, a string, tuple, integer are all value data types and lists and dictionaries are reference (if data is mutable it is reference). Value type data makes a copy of the data and reference type makes a copy of the reference to the values. Using references to the data is more efficient than making a copy of the data because you are just copying the reference instead of potentially gb’s of data – assuming you’re using a large data file.

# Variable Scope, Docstring and Classes:

Variable scope, otherwise known as encapsulation allows programmers to access data from other parts of your program through global variables. Scope represents different parts of your code that are separate from each other ie. each function has its own scope. If a program has four functions within it, there are five scopes within the program, one for each function and the last for the global scope. A variable you set in the global scope is called a global variable and any variable set in a function is called a local variable. Global variables are allowed to be accessed within the child functions unless there is a variable with the same name inside of the function. Functions can’t use the variables from other functions. It is like a one-way mirror.

Docstrings are a summary or explanation of what the function is doing and includes what arguments and results occur if there are any. This allows for other programmers to get a better understanding of your intent with each function. The formatting will largely depend on the environment you’re working in. You can access the docstring within the tooltip or ‘command / control I’ within your IDE. A tip when updating and maintaining code is to update your docstring when you revise your program so that it can stay up to date with the processing of your program.

Classes are a way of grouping functions and in our use case this week, it was mostly intended for us to be able to differentiate our code within the separations of concerns between, Data, Processing and Presentation. We will learn more in week eight but for this week we were supposed to use @staticmethod.

# Assignment:

To put our learnings into practice we refactored some code from week five and instead of using the while loop to perform all of the operations we needed to move some of the code blocks into their own functions. We needed to put our functions into different category types, file processing, data processing and i/o processing. It was a bit daunting to begin with because the code we were supposed to update was not ours, so it took some time getting acquainted with, even if it was essentially the same code as we wrote. The program was the CD Inventory menu where a user can load, add, display, delete, and save lines of their inventory from memory. Example starter code for adding CD Inventory to memory below:

**elif** strChoice == 'a':

# 3.3.1 Ask user for new ID, CD Title and Artist

# TODO move IO code into function

strID = input('Enter ID: ').strip()

strTitle = input('What is the CD\'s title? ').strip()

stArtist = input('What is the Artist\'s name? ').strip()

# 3.3.2 Add item to the table

# TODO move processing code into function

intID = int(strID)

dicRow = {'ID': intID, 'Title': strTitle, 'Artist': stArtist}

lstTbl.append(dicRow)

IO.show\_inventory(lstTbl)

**continue** # start loop back at top.

The solution I came up with to ensure both of these components were in their own functions and then called in their respective place is below:

@staticmethod

**def** **user\_input**():

"""Function to get user input for CD inventory to put into memory

Arg: none

Return: list of user inputs

"""

strID = input('Enter ID: ').strip()

strTitle = input('What is the CD\'s title? ').strip()

stArtist = input('What is the Artist\'s name? ').strip()

user\_lst = [strID, strTitle, stArtist]

**return** user\_lst

@staticmethod

**def** **user\_input\_proc**():

"""

Function to put user data into a list of dictionaries

Args:

None.

Returns

None.

"""

user\_lst = IO.user\_input()

intID = int(user\_lst[0])

dicRow = {'ID': intID, 'Title': user\_lst[1], 'Artist': user\_lst[2]}

lstTbl.append(dicRow)

**return**

Call to the function in the elif statement of the while loop:

**elif** strChoice == 'a':

# 3.3.1 Ask user for new ID, CD Title and Artist

# 3.3.2 Add item to the table

DataProcessor.user\_input\_proc()

IO.show\_inventory(lstTbl)

**continue** # start loop back at top.

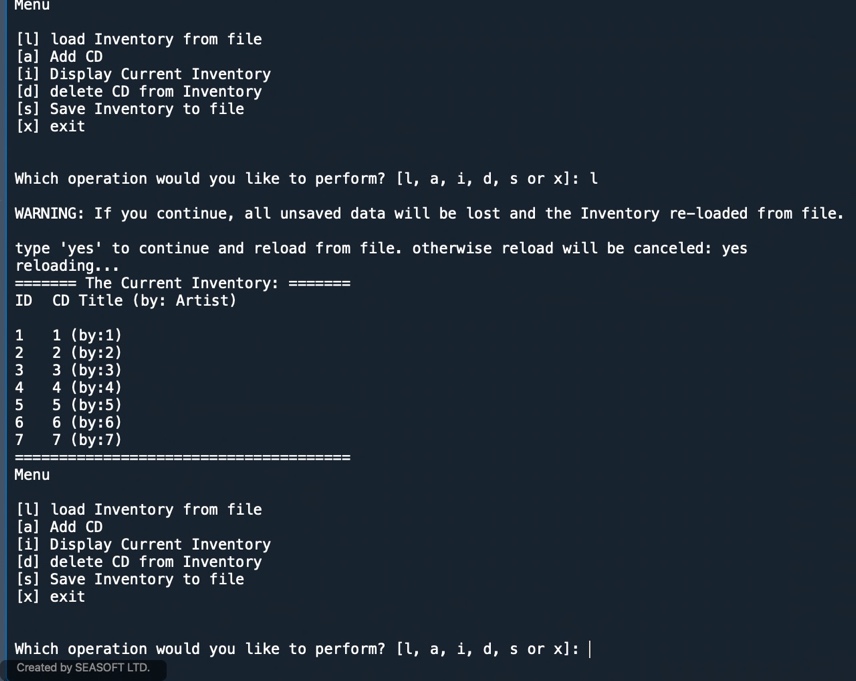
As you can see in the example above, the process to ensure our code to work was heavily dependent on each of the two functions, because you needed to add the user input to memory and then the processing function needed to call that function to add the data into the dictionary item by item and then appended to a list. 

Figure 2 - Spyder Results for loading inventory

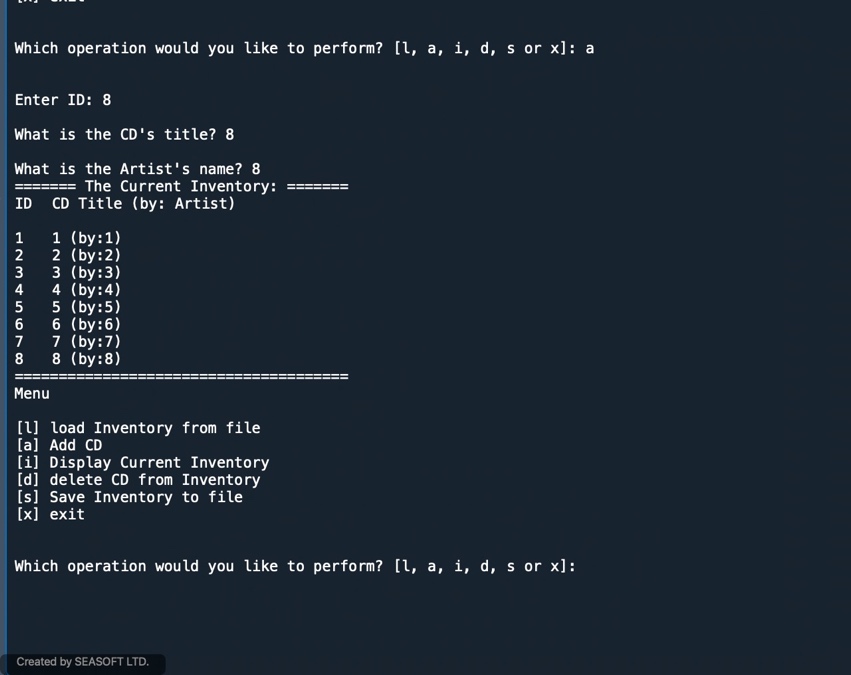


Figure 3 - Spyder Results for adding inventory

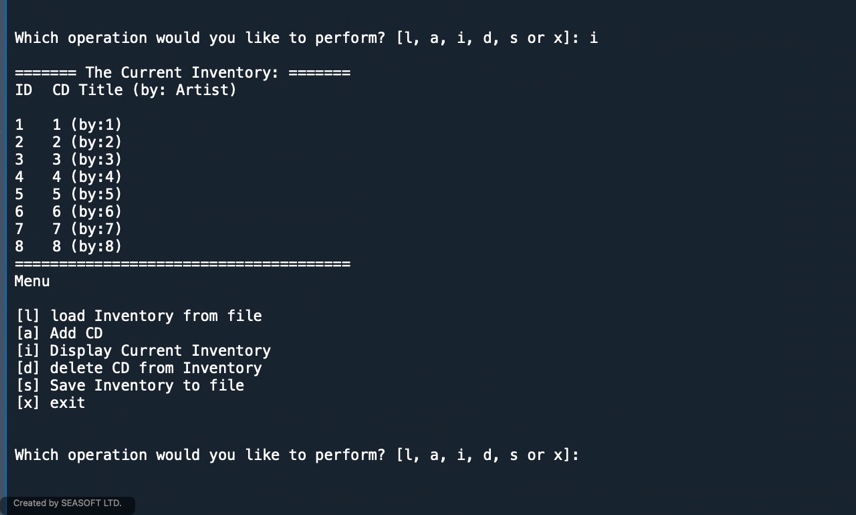
To continue the program, we needed to add each of the main items with the exception of the load component into its own function. Results showing success of each part:

Figure 4 - Spyder Results - displaying inventory

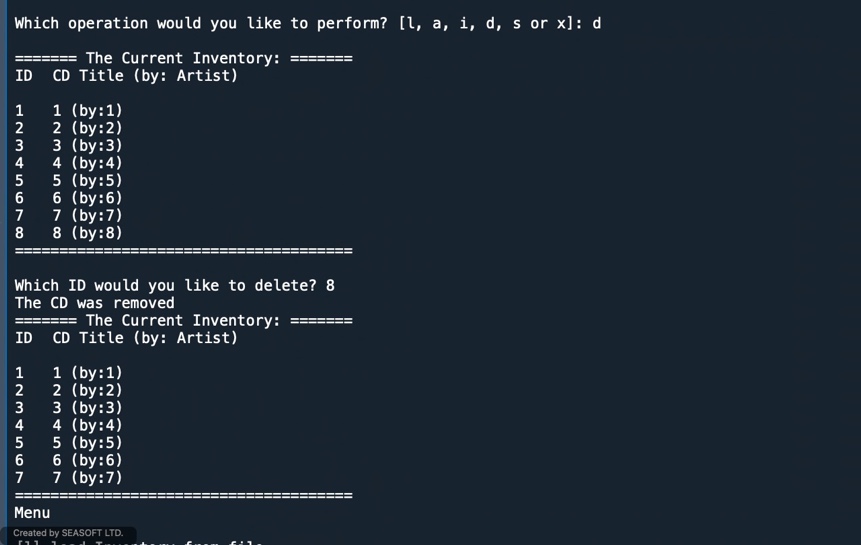


Figure 5 - Spyder Results deleting rows

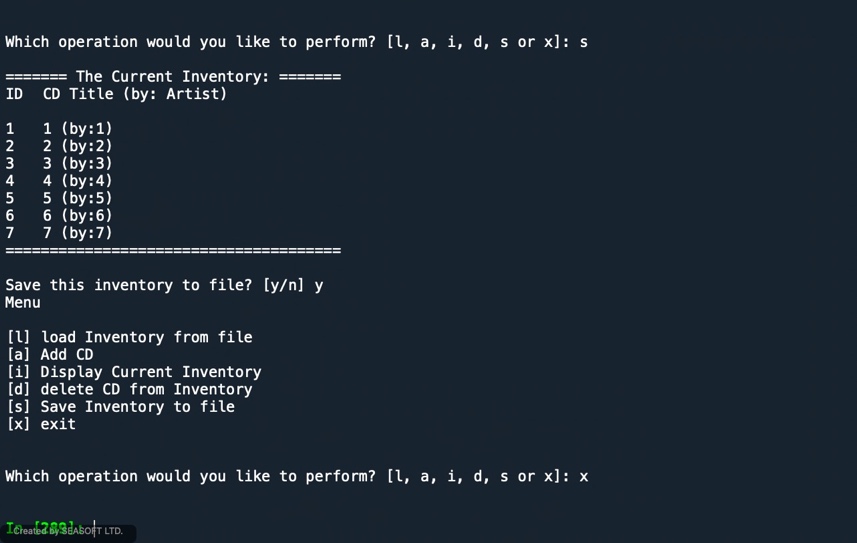


Figure 6 - Spyder Results saving and exiting program



Figure 7 - Spyder Results .txt file

As seen in the above screencaps, the updates made to each of the sections that were moved to their own functions are working correctly. I ran into a little bit of a snag that I needed some help with in regards to writing to a file, I didn’t realize I needed to change lstTbl to table in the code because of the argument and kept on overlooking that simple update and was thinking there was something incorrect elsewhere. One lesson I’m learning is that often times when I am stuck, I go down a rabbit hole and my stubbornness can get the best of me in trying to solve the problem. I should have taken some time away and gone for a walk and asked the question earlier. Additionally, I had to include another while loop to account for the File Processor to work if there was no file to load. That said, it was a good learning and each week there is a little item that this happens to me on which is good to see because it means I am learning the material even if at times it can be difficult.

# Conclusion:

During the week we learned a lot of new topics about Functions and how we can use functions to separate our code into different logical areas. We learned about named and positional arguments (or parameters and python is implicit with its use of a positional argument, you can even mix the two styles within your result but there is a specific format that is needed). We also learned that a function can have zero (or None), one or multiple result values. We also learned about Variable scope and how variables are handled within and outside of functions. Additionally, we learned about docstrings and their benefit to bringing clarity to your functions within your program. And finally, we were used our learnings to refactor our week five sample code to get it working by placing various parts of the while loop into their own specific functions and then calling them back in the while loop.

# Appendix:

# 

Figure 8 - Terminal Results pt. 1

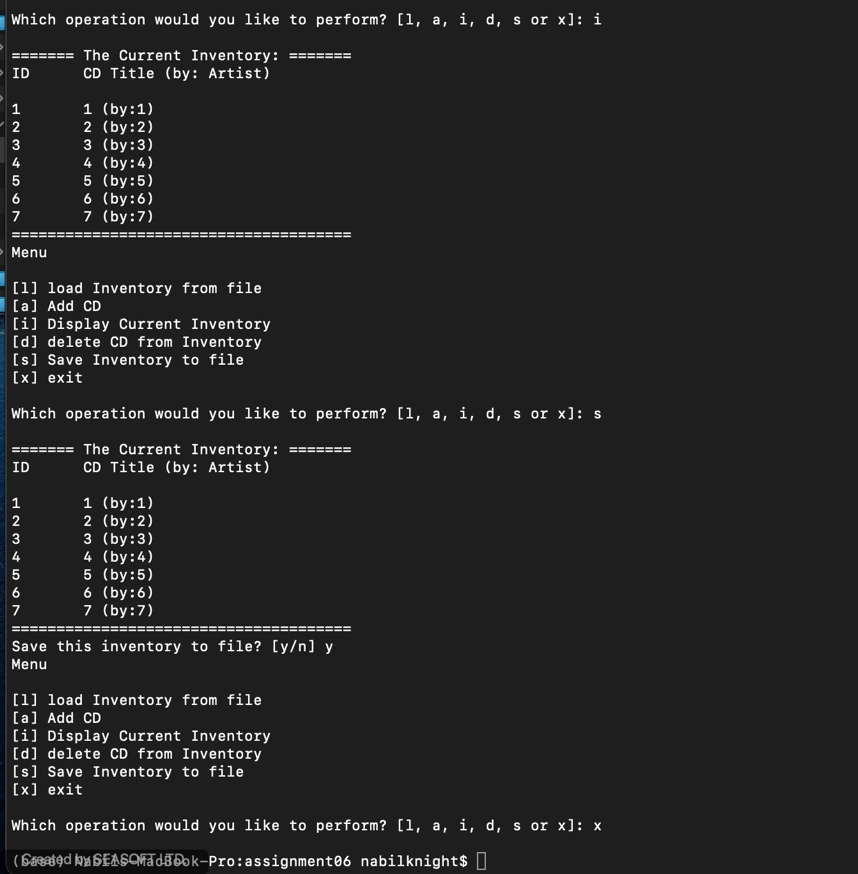


Figure 9 - Terminal Results pt. 2