Rei Bengu
August 13, 2021
IT FDN 110 B
Assignment 06

Functions and Classes

Introduction

In this module we take a leap at learning how to be efficient and organized for programs that can be expanded over time or are just long and involved to begin with. The focus is specifically on classes and functions. A function is just like any other function we have been using thus far, however in this context it is user-defined. A class is how we can organize functions with a common goal. For example, a set of math functions can be structured under a single class. The benefit to utilizing classes and user-defined functions instead of just coding an entire program normally is that it allows for better separation of priorities (Data, Processing, Input/Output) as well as makes the programmers life easier by allowing code to be referenced instead of copy and pasted, reducing the amount of time required for rework and lowering the possibility of mistakes. The script and this knowledge document can be found at github to see the progression of this assignment.

CD Inventory Script

To put to practice the skills learned in this module, the basic script created and refined over past modules which asked the user to enter basic CD data to create an expanding table of information was revisited and updated with the concepts learned in this module on functions and classes.

Our code which previously went from Data to a mixture of interwoven Processing and Input/Output, is now able to be shuffled around to utilize separation of priorities and flow from Data to Processing to Input/Ouput thanks to functions. The data section which starts off the script is unchanged, shown below for context, and initializes some of the more common variables that are used throughout the program in its processing. These variables are initialized to allow multiple programmers to have a reference list of what variables are being used and what they represent, to ensure each programmer uses the exact same syntax instead of having to dig through the code to find what they are looking for. The variables are also empty when initialized but set to the type of data they will be used as later, i.e strChoice = '' establishes a variable called 'strChoice' that contains an empty string.

```
# -- DATA -- #
strChoice = '' # User input
lstTbl = []  # list of lists to hold data
dicRow = {}  # list of data row
strFileName = 'CDInventory.txt'  # data storage file
objFile = None  # file object
```

After the script declares some initial variables used throughout the code it launches into the next phase of the code, Processing. Here the code creates all of the functions organized by classes which can be utilized throughout the program itself. There are many classes and functions within this program, but since the actual contents of the code itself is relatively unchanged from the previous iteration, only one example of a class will be shown which utilizes all changes made throughout the processing section, the same format is used throughout to create all the classes and functions. In our example below, a class titled 'DataProcessor'. This class is used to process data during runtime, and contains within it two functions; 'add_cd()' and 'del_cd()'. The first function, add_cd() adds an entry to the data structure, the second function del_cd() deletes an entry from the data structure. Thanks to the docstrings (everything contained between the

“”” “””) each function is explained simply to the programmer, from what parameters are to be passed on into the function, what type of output to expect, etc.

The add_cd() function uses 3 arguments; strID, strTitle and stArtist. These 3 arguments are explained within the docstring of the function itself, but when the function is called upon later in the program, the user can fill in these arguments witch are then passed along to be consumed within this function. The code itself that actually achieves the adding/deletion of entries from the table is similar to previous modules.

```
1. # -- PROCESSING -- #
2. class DataProcessor:
3.     """ Processes data during runtime """
4.     @staticmethod
5.     def add_cd(strID, strTitle, stArtist):
6.         """ Adds CD to data structure (list of dicts)
7.
8.         Args:
9.             strID (string): ID for new entry
10.
11.                 strTitle (string): Title of CD to be added
12.
13.                 stArtist (string): Artist of CD to be added
14.
15.             Returns:
16.                 None.
17.             """
18.
19.             intID = int(strID)
20.             dicRow = {'ID': intID, 'Title': strTitle, 'Artist': stArtist}
21.             lstTbl.append(dicRow)
22.
23.         @staticmethod
24.         def del_cd(intIDDel, table):
25.             """ Deletes CD from data structure (list of dicts)
26.
27.             Args:
28.                 intIDDel (int): ID of entry to be deleted
29.
30.                 table (list of dict): 2D data structure (list of dicts)
   that holds the data during runtime
31.
32.             Returns:
33.                 None.
34.             """
35.
36.             intRowNr = -1
37.             blnCDRemoved = False
38.             for row in table:
39.                 intRowNr += 1
40.                 if row['ID'] == intIDDel:
41.                     del table[intRowNr]
```

```
                    blnCDRemoved = True
                    break
            if blnCDRemoved:
                print('The CD was removed')
            else:
                print('Could not find this CD!')
```

Moving on to the last section of the script, the Input/Output, or Presentation section. This section contains all of the actual interactions with the user, and calls upon the functions laid out within the Processing section to carry out any actual data processing. The main structure is again, similar to that of previous modules: a main while loop that allows the user to infinitely cycle through their menu options, and add/remove/view/read/write entries. Below is an example of the add_cd() function being used in action, it is nested within the aforementioned while loop.

```
    elif strChoice == 'a':
        # 3.3.1 Ask user for new ID, CD Title and Artist
        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
        DataProcessor.add_cd(strID, strTitle, stArtist)
        continue  # start loop back at top.
```

Since the add_cd() function is contained within the DataProcessor class, it is called upon as DataProcessor.add_cd(). The script first asks the user for inputs of ID, Title, Artist as per usual, but then instead of doing the processing in the subsequent lines of code, it passes those results as arguments within the DataProcessor.add_cd() function, which acts as Python copying and pasting the function being called upon into that line. This makes it so that the Input/Output section does not get bogged down with processing, and instead focuses solely on what its name suggests.

Figure 2 below shows the CDInventory.txt file with some pre-filled entries before the code is ran. An example of the output of this script can be seen below in in **Error! Reference source not found.** which is directly from the console of the Spyder IDE, going through every menu choice available as well as in which is directly from a terminal. **Error! Reference source not found.** shows the CDInventory.txt file after the program is finished running, note the capitalized 3rd row which can be seen being added in Figure 2. Figure 4 shows an example of the script running in a terminal.

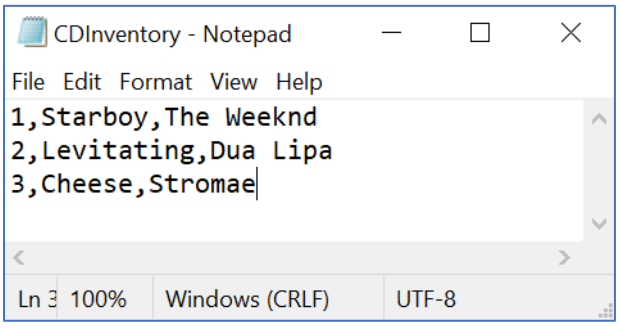

Figure 1 - Initial 'CDInventory.txt' with pre-filled values.

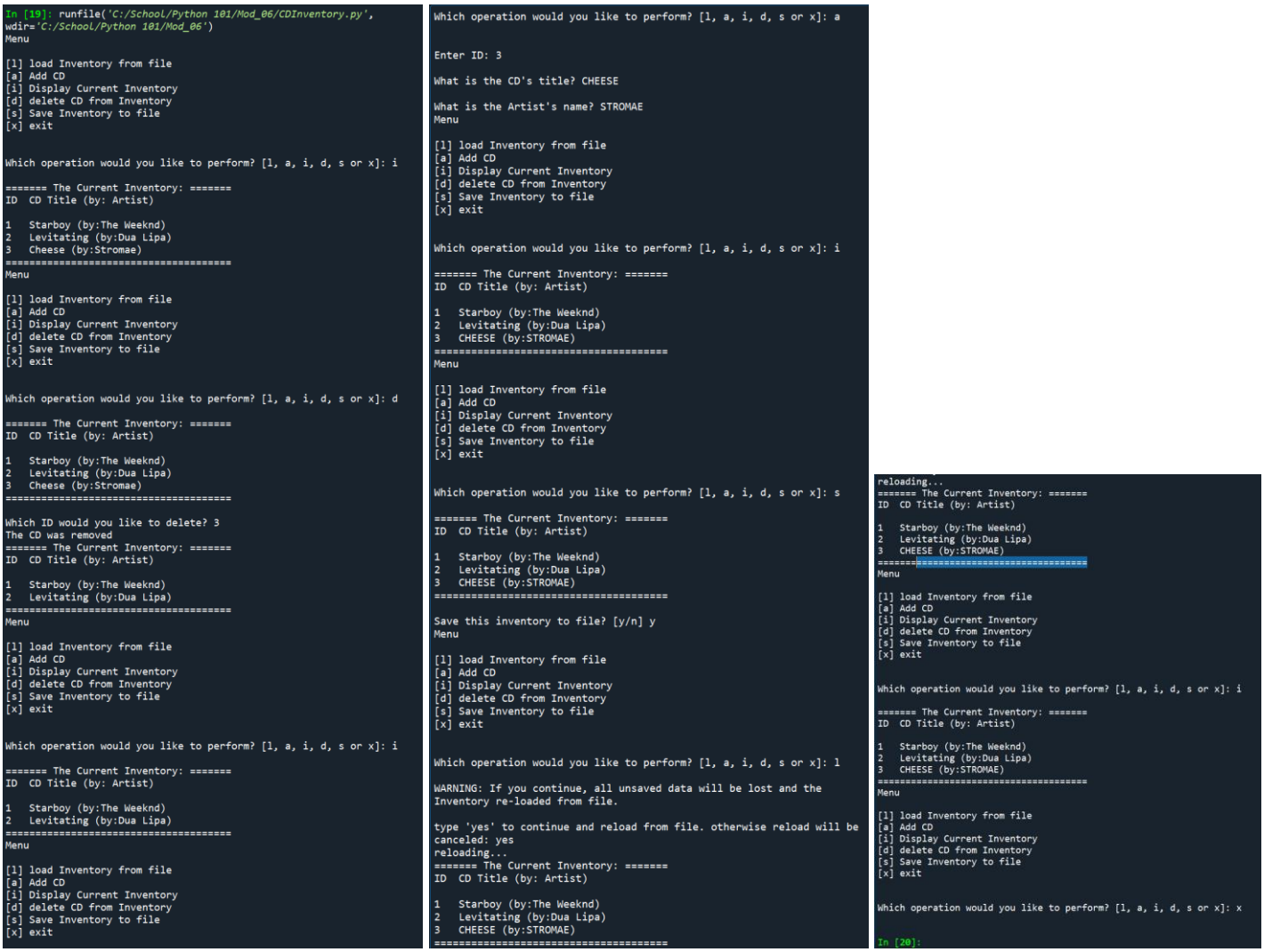

Figure 2 – Example output from working script in Spyder IDE

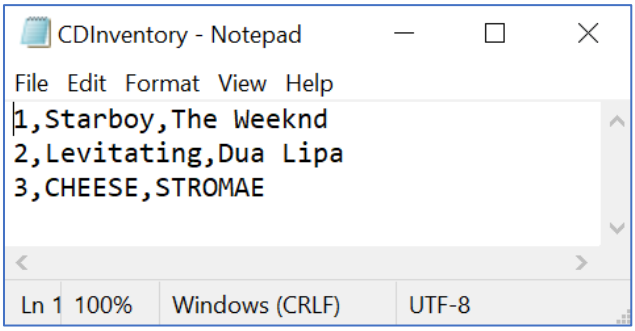

Figure 3 - CDInventory.txt generated by the script after it was run in Spyder IDE

Figure 4 - Example of the program running in a terminal.

Summary

Throughout this module, we focused on refining what we have learned in the past to make more efficient programs. Using functions, parameters, classes in order to centralize sections of a programs code. As the CDInventory has grown through each iteration, the size and scope have grown as well, and utilizing functions to streamline the program has already addressed one of the main concerns I had with the repetitive nature of the program offering more margin for error as changes were made that needed to be propagated throughout.