Ann Allen

Introduction to Programming – Python

Assignment 6

August 15, 2020

Classes and Functions

Introduction

In Assignment 6, the focus is on using functions effectively in a program. These functions all exist withing a set of three classes that were predetermined for the program. Combined, the program CDInventory.py script demonstrates a framework for using classes and functions correctly and effectively within a program.

GitHub Repository

This document and CDInventory.py can be found on GitHub at:
https://github.com/annspiral/Assignment_06

Classes

Classes create groupings or categories to make it easier to use functions. HackerEarth put it succinctly saying, "Programmers use classes to keep related things together."[1] Rather than having one, complete set of functions, classes create branches of similar categories.

Classes and Separation of Concerns

Module 6 only touches on using classes within a program. The starter script in Assignment 6 provided a class structure already predetermined. This structure consisted of three classes: DataProcessor, FileProcessor and IO. These classes closely mirror our previously discussed program ideal of Separation of Concerns. In Module 5 we learned to separate our programs into Data, Processing and Presentation, using these three areas as sections for our script template[2]. In Module 6, two of these three areas are represented by classes:

Processing → FileProcessor, DataProcessor classes
Presentation → IO class

[1] HackerEarth, https://www.hackerearth.com/practice/python/object-oriented-programming/classes-and-objects-i/tutorial/, August 17, 2020
[2] Biesinger, Dirk. Introduction to Programming – Python, FDN_Py_Module_05.pdf, August, 15, 2020

Each of the functions added in CDInventory.py for Assignment 6 went into one of these three classes, organizing the program code in logical groupings.

get_new_entry() was added to the IO class
add_CD() was added to the DataProcessor class
delete_CD() was added to the DataProcessor class
get_sorted_values_list() was added to the DataProcessor class
write_file() was added to the FileProcessor class

Using Classes

To use the classes in CDInventory.py, each function call was preceded by the name of the class the function resides in. Listing 1 shows the code for adding a new CD. This code has been reduced to four lines of code using 2 classes. The classes make clear sense here. The first line of code gets a sorted list of the table IDs, using the DataProcessor class to process the table data. The return value of DataProcessor.get_sorted_values_list() is stored in a variable, lstTestIDs. This list variable is then passed into the function, IO.get_new_entry(), to ask the user for the new CD information. The sorted list of values, lets the get_new_entry() function assign a unique ID value for the new entry. IO.get_new_entry() returns a tuple value that contains the three string values for the new entry. This tuple is then passed to the next function call. The third line uses a DataProcessor class function, add_CD(), to add the CD info to the inventory. Lastly, the fourth line returns to the IO class to confirm the addition to the user.

```
        # 3.3 process add a CD
        elif strChoice == 'a':
            # 3.3.0 get the sorted list of IDs currently in use
            lstTestIDs = DataProcessor.get_sorted_values_list(lstTbl)
            # 3.3.1 Ask user for new ID, CD Title and Artist
            tplUserEntry = IO.get_new_entry(lstTestIDs)
            # 3.3.2 Add item to the table
            # 3.3.2.1 Add item with users values to the table
            DataProcessor.add_CD(tplUserEntry, lstTbl)
            # 3.3.2.2 Display inventory to user to confirm CD added
            IO.show_inventory(lstTbl)
            continue  # start loop back at top.
```

Listing 1

The code in Listing 1 also demonstrates many of the features of functions. The functions take different parameters and, in some cases, return values. More of the details on functions are covered in the next section.

Functions

Most of the effort in Assignment 6 went into defining and using functions. In some cases, I was able to simply move the existing code to a function with limited changes.

Return Values
Functions can return nothing or return any number of values (although multiple values are returned as a tuple)[3]. Another way to return multiple values is to return a tuple directly. In one case, I wanted my function, get_sorted_values_list(), to return a full set of IDs already used in the inventory. I planned to return a tuple filled with the IDs in the inventory. I also wanted to sort the tuple to easily retrieve the highest ID value in order to set a new, unique ID. The sort method only exits on the list type, since it is mutable. I have read about a sort option for tuples[4], but it is currently not in the scope of what our class has covered. Instead, I chose to return the set of IDs as a list.

It is helpful to note that functions always end when they reach a return value.[5] This gives you a way to break out of the function with different return value options, although I did not use this in CDInventory.py.

Reference Values
Functions can also modify reference variables in place, which does not require making a copy and saves memory. The function can work directly on the table variable passed as a parameter[6]. In CDInventory.py, many of the functions take the inventory table as a function argument. The inventory table is a 2D list of dictionaries. The DataProcessor functions used to add or delete a CD from the inventory, `add_CD()` and `delete_CD()`, both edit the list of dictionaries directly and have no return value (CDInventory.py listing lines 25-47, lines 51-78).

Parameters/Arguments
Functions can take a variety of different parameters: no parameter, many parameters, required parameters, default parameters, value or collection types. These parameters can also be referred to as arguments. These two terms are used interchangeably[7].

Multiple parameters
The `add_CD` function in the DataProcessor class takes two parameters: `tplEntry` and `table` (Listing 2).

```
25.     def add_CD(tplEntry, table):
```

[3] Biesinger, Dirk, Introduction to Programming – Python, Module 6 Video 2, https://www.youtube.com/watch?v=bcOdd8J5vlM&feature=youtu.be, Aug. 15, 2020
When including multiple return variables in a function, these are converted automatically by Python into a tuple when passed. But there are other ways to return multiple return values: returning a list, returning a tuple directly, returning a dictionary or returning an object.
GeeksforGeeks, https://www.geeksforgeeks.org/g-fact-41-multiple-return-values-in-python/, Aug. 17, 2020
[4] Gitconnected, https://levelup.gitconnected.com/sort-elements-in-python-817a0c2b810b, Aug. 17, 2020
[5] Dawson, Michael. Python Programming, 3rd Edition, Course Technology, 2010, Page 164
[6] Biesinger, Dirk, Introduction to Programming – Python, Module 6, Video 3, https://www.youtube.com/watch?v=pTr4QqSQXOU, August 15, 2020
[7] Biesinger, Dirk, Introduction to Programming – Python, FDN_Py_Module_06.pdf, August 15, 2020

Listing 2: Function definition for DataProcessor.add_CD() function in CDInventory.py

The parameter, `tplEntry`, is a tuple with three string values that correspond to the ID, Title and Artist of the CD. The second parameter, table, is the 2D list of dictionaries table that represents the CD inventory. The details of these parameters are outlined in the function docstring. The docstring is described more in the next section.

No parameters

A function can also take no parameters at all. The IO class function, `print_menu()` is an example of a function that takes no parameters. This function uses the `print()` method to display menu options for the user. Since this menu is redisplayed throughout the program, it is useful to have a single function call to display the menu again each time. To call the function, the programmer uses the empty parentheses as seen on line 282 of CDInventory.py.

Default parameters

Functions can also have default parameters. If a parameter is set to equal a value in the function definition, this value will be used if the parameter is omitted when called. I used this construct for the function get_sorted_value_list().

Listing 3 shows that get_sorted_value_list() has two parameters, table and key.

```
    def get_sorted_values_list(table, key='ID'):
        """Function to get all values of the specified key from a list of

            """
            lstIDsUsed = []
            #get a list of all values for the key parameter
            for row in lstTbl:
                lstIDsUsed.append(row.get(key))
            lstIDsUsed.sort()
            return lstIDsUsed

```

Listing 3: CDInventory.py code show a function with a default parameter

The key parameter is set equal to the value “ID”. If no second parameter is passed in, the function will use the default value of “ID”. This is the value I wanted to use for this case, but I wanted to make the function more flexible to consider using it to make a list of artist or albums in the future. Calling this function on line 303, I only need to pass in the table (Listing 4).

```
303.            lstTestIDs = DataProcessor.get_sorted_value_list(lstTbl)
```

Listing 4: Calling a function without one of the parameters

Programmers need to take care when using default parameters. As soon as one parameter in a function is set to a default, any following parameters must also have defaults. Python will expect defaults for any additional parameters.[8]

DocStrings

`docstrings` are full featured comments used to document functions. These comments include special formatting that is easy to read in the code and follow a convention that can be used by other programs to pull out the content to display or document the function[9]. Docstrings are identified by triple pound signs at the beginning and the end – “”” “””. The format is defined as:

```
“””[short description]
        [empty line]
[full description]
        [empty line]
[Args:] or [Parameters:]
        [each argument with colon and description (or None)]]
        [empty line]
[Returns:]
        [each return with colon and description (or None)]]
        [empty line]
“””
```

Listing 5 shows an example of a docstring for the function `add_CD()`.

```python
    def add_CD(tplEntry, table):
        """Function to add a CD dictionary entry to the table/inventory

        Takes a tuple of entry strings (ID, Title, Artist) and creates a
        dictionary collection {'ID':num, 'Title':'string, 'Artist':string}
        that is added to the full inventory table, which is a list of dictionaries

        Args:
            tplEntry: a tuple of strings containing values to be added into
            dictionary collection and table - expected order of ID, Title, Artist.
            table (list of dict): 2D data structure (list of dicts) that holds
            the data during runtime

        Returns:
            None

        """
```

[8] Biesinger, Dirk, Introduction to Programming – Python, Module 6, Video 2, https://www.youtube.com/watch?v=bcOdd8J5vlM&feature=youtu.be, August 15, 2020
[9] Biesinger, Dirk, Introduction to Programming – Python, Module 6, Video 4, https://www.youtube.com/watch?v=bMOVdz9B-Vk, August 15, 2020

Listing 5: Example of docstring for function add_CD in CDInventory.py

When commenting the functions in CDInventory.py, I found that I often wanted to make my short descriptions too long. I returned to shorten them a few times to ensure the short description was brief and the full description contained the necessary details.

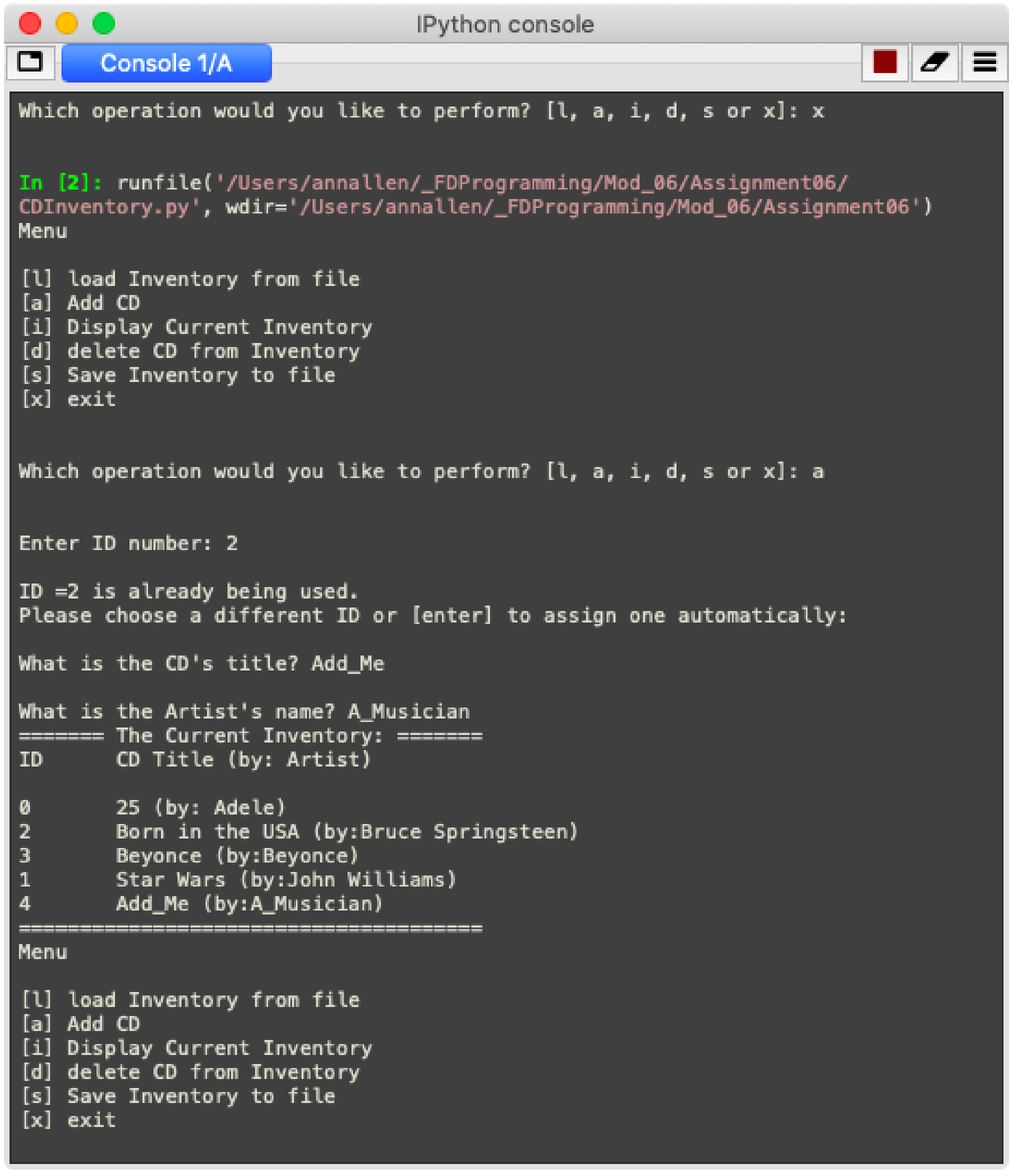
Spacing

A last function convention to note is that functions in Python are expected to be separated by 2 spaces in the code.[10]

Examples of CDInventory.py script

CDInventory.py running in Spyder IDE

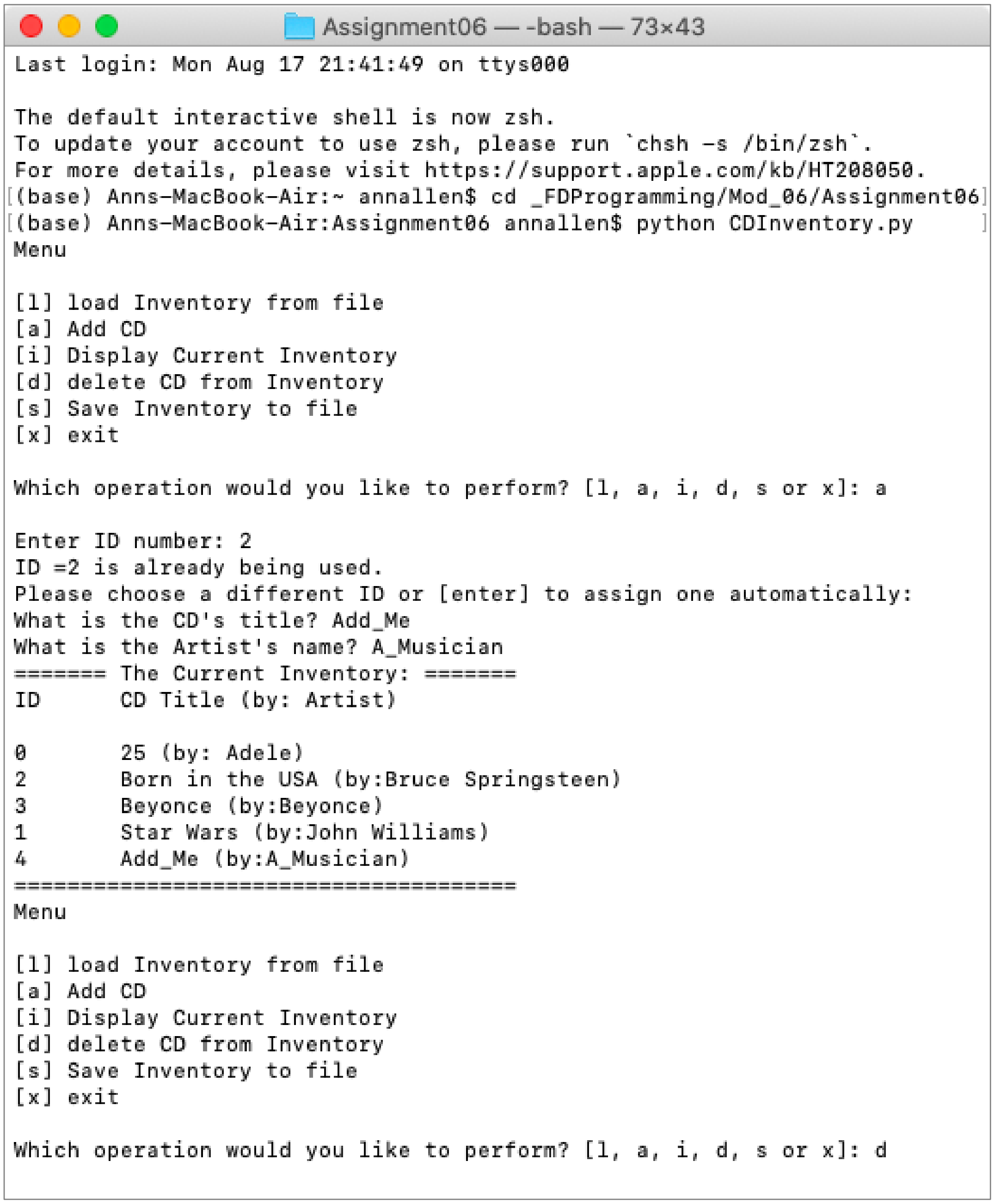
[10] London App Developer, https://www.youtube.com/watch?v=_ypAw_pCOt8&feature=youtu.be, August 17, 2020



```
Which operation would you like to perform? [l, a, i, d, s or x]: x


In [2]: runfile('/Users/annallen/_FDProgramming/Mod_06/Assignment06/
CDInventory.py', wdir='/Users/annallen/_FDProgramming/Mod_06/Assignment06')
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit


Which operation would you like to perform? [l, a, i, d, s or x]: a


Enter ID number: 2

ID =2 is already being used.
Please choose a different ID or [enter] to assign one automatically:

What is the CD's title? Add_Me

What is the Artist's name? A_Musician
======= The Current Inventory: =======
ID      CD Title (by: Artist)

0       25 (by: Adele)
2       Born in the USA (by:Bruce Springsteen)
3       Beyonce (by:Beyonce)
1       Star Wars (by:John Williams)
4       Add_Me (by:A_Musician)
======================================
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit
```

Figure 1: Using CDInventory.py in the Spyder IDE to add a CD

CDInventory.py running in MacOS Bash terminal



```
Last login: Mon Aug 17 21:41:49 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
[(base) Anns-MacBook-Air:~ annallen$ cd _FDProgramming/Mod_06/Assignment06]
[(base) Anns-MacBook-Air:Assignment06 annallen$ python CDInventory.py      ]
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [l, a, i, d, s or x]: a

Enter ID number: 2
ID =2 is already being used.
Please choose a different ID or [enter] to assign one automatically:
What is the CD's title? Add_Me
What is the Artist's name? A_Musician
======= The Current Inventory: =======
ID      CD Title (by: Artist)

0       25 (by: Adele)
2       Born in the USA (by:Bruce Springsteen)
3       Beyonce (by:Beyonce)
1       Star Wars (by:John Williams)
4       Add_Me (by:A_Musician)
======================================
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [l, a, i, d, s or x]: d
```

Figure 2: Using CDInventory.py in the MacOS Bash terminal to add a CD

Summary

With Classes and functions implemented, the final body of the program for CDInventory.py reads like pseudocode, line by line. This makes the program extremely clear and easy to understand. As any issues arise, I could then look into an individual function. Each function required a significant amount of documentation in the docstring, which was a time consuming task. As the program grows, this time should pay off to allow for clear debugging and reuse. When implementing functions, I also found that I spent more time considering the right parameters and return values than the time to code the function itself. Planning and documenting took on stronger roles in Assignment 6.

Appendix

CDinventory.py

```python
#------------------------------------------#
# Title: Assignment06_Starter.py
# Desc: Working with classes and functions.
# Change Log: (Who, When, What)
# DBiesinger, 2030-Jan-01, Created File
# AAllen, 2020-Aug-15, Added function get_new_entry() in IO
# AAllen, 2020-Aug-15, Added functions add_CD(), delete_CD() in DataProcessor
# AAllen, 2020-Aug-15, Added function write_file() in FileProcessor
# AAllen, 2020-Aug-15, updated function comments, added get_sorted_values_list()
# AAllen, 2020-Aug-15, added error checking in get_new_entry() for ID
# AAllen, 2020-Aug-17, cleaned up some docstrings
# AAllen, 2020-Aug-18, moved call to DataProcessor.get_sorted_values_list()
#                        out of IO.get_new_entry()
#------------------------------------------#

# -- 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


# -- PROCESSING -- #
class DataProcessor:
    @staticmethod
    def add_CD(tplEntry, table):
        """Function to add a CD dictionary entry to the table/inventory

        Takes a tuple of entry strings (ID, Title, Artist) and creates a
        dictionary collection {'ID':num, 'Title':string, 'Artist':string}
        that is added to the full inventory table, which is a list of dictionaries

        Args:
            tplEntry: a tuple of strings containing values to be added into
            dictionary collection and table - expected order of ID, Title, Artist.
            table (list of dict): 2D data structure (list of dicts) that holds
            the data during runtime

        Returns:
            None

        """
        # convert tuple values into dictionary item
        dicRow = {'ID': int(tplEntry[0]),
                  'Title': tplEntry[1],
                  'Artist': tplEntry[2]}
        # add dictionary to current table
        table.append(dicRow)


    @staticmethod
    def delete_CD(intDelID, table):
        """Function to delete a CD dictionary entry from the table/inventory

        Deletes the row identified by the ID value from the inventory table.
        The row is a dictionary entry that represents each CD.

        Args:
            intDelID: ID value for the dictionary collection in the list that
            will be deleted
            table (list of dict): 2D data structure (list of dicts) that holds
            the CD inventory data during runtime

        Returns:
            None.

        """
        intRowNr = -1
        blnCDRemoved = False
        for row in table:
            intRowNr += 1
            if row['ID'] == intDelID:
                del table[intRowNr]
                blnCDRemoved = True
                break
        if blnCDRemoved:
            print('The CD was removed')
        else:
            print('Could not find this CD!')


    @staticmethod
    def get_sorted_values_list(table, key='ID'):
        """Function to get all values of the specified key from a list of
        dictionaries

        Creates a sorted list of all the values for the key provided that are
        currently in the inventory table (2D list of dictionaries)

        Args:
            table: expects a 2D table that is a list of dictionaries
            key: key for which values are retrieved from the dictionaries,
            defaults to look for a key name 'ID'

        Returns:
            sorted list of all values with the provided key that are
            in the 2D table

            """
            lstIDsUsed = []
            #get a list of all values for the key parameter
            for row in lstTbl:
                lstIDsUsed.append(row.get(key))
            lstIDsUsed.sort()
            return lstIDsUsed


class FileProcessor:
    """Processing the data to and from text file"""

    @staticmethod
    def read_file(file_name, table):
            """Function to manage data ingestion from file to a list of dictionaries

            Reads the data from file identified by file_name into a 2D table
            (list of dicts) table one line in the file represents one
            dictionary row in table.

            Args:
                file_name (string): name of file used to read the data from
                table (list of dict): 2D data structure (list of dicts) that holds
                the data during runtime

            Returns:
                None.
            """
            table.clear()  # this clears existing data and allows to load data from file
            objFile = open(file_name, 'r')
            for line in objFile:
                data = line.strip().split(',')
                dicRow = {'ID': int(data[0]), 'Title': data[1], 'Artist': data[2]}
                table.append(dicRow)
            objFile.close()
            pass


    @staticmethod
    def write_file(file_name, table):
            """Function to write updated inventory list of dictionaries to file

            Writes the data in table to the file identified by file_name. Table is
            a list of dictionaries. Each dictionary item is written to a line in
            the file, with a newline ending. The dictionary is written as the
            value from each key/value pair separated by a comma

            Args:
                file_name (string): name of file used to save to
                table (list of dict): 2D data structure (list of dicts) that holds
                the data during runtime

            Returns:
                None
            """
            objFile = open(strFileName, 'w')
            # for each dictionary 'row' in the table, convert values into a single,
            # comma separated string with an ending newline and write to file
            for row in table:
                lstValues = list(row.values())
                # ID integer value needs to be converted to string for output
                lstValues[0] = str(lstValues[0])
                objFile.write(','.join(lstValues) + '\n')
            objFile.close()
            pass

# -- PRESENTATION (Input/Output) -- #

class IO:
        """Handling Input / Output"""

    @staticmethod
    def print_menu():
```

```python
        """Displays a menu of choices to the user

        Args:
            None.

        Returns:
            None.
        """

        print('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
        print('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')


    @staticmethod
    def menu_choice():
        """Gets user input for menu selection

        Args:
            None.

        Returns:
            choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x

        """
        choice = ' '
        while choice not in ['l', 'a', 'i', 'd', 's', 'x']:
            choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
        print()  # Add extra space for layout
        return choice


    @staticmethod
    def show_inventory(table):
        """Displays current inventory table


        Args:
            table (list of dict): 2D data structure (list of dicts) that
            holds the data during runtime.

        Returns:
            None.

        """
        print('======= The Current Inventory: =======')
        print('ID\tCD Title (by: Artist)\n')
        for row in table:
            print('{}\t{} (by:{})'.format(*row.values()))
        print('======================================')


    @staticmethod
    def get_new_entry(lstTestIDs):
        """Requests the new entry values, ID, Title, Artist

         Args:
            lstTestIDs: a list containing IDs currently in use in the table

        Returns:
            tplNewEntry: a tuple of three strings with the user's input for the
            (ID, Title, Artist) for a new CD entry

        """
        # Get a unique entry ID
        entryID = '' # create empty default ID
        validID = False # Flag to indicate valid ID identified
        # request an ID number from user
        entryID = input('Enter ID number: ').strip()
        # verify ID number and re-ask as needed
        while not validID:
            # if the user does not enter an ID, find one and assign it
            # if the list of current IDs is empty, start with ID 0
            if entryID == '' and ((lstTestIDs == []) or (lstTestIDs == None)):
                entryID = 0
                break
            # assign a CD ID, by finding the next ID number to use
            # get last, highest ID value and get next ID
            elif entryID == '':
                entryID = lstTestIDs[-1] + 1
                break
            # if the user enters a non-numeric character, ask for ID again,
            # and loop back through
            if not entryID.isnumeric():
                entryID = input('Please Enter a numerical ID number: ').strip()
                continue
            # if the user enters an ID already in the list, ask for a new ID
            if int(entryID) in lstTestIDs:
                entryID = input('ID =' + entryID + ' is already being used.\n' +
                                'Please choose a different ID or [enter] to' +
                                ' assign one automatically: ').strip()
            else:
                validID = True

        # get the entry title and artist, these can be blank
        entryTitle = input('What is the CD\'s title? ').strip()
        entryArtist = input('What is the Artist\'s name? ').strip()

        # create new entry tuple with ID, Title and Artist
        tplNewEntry = (entryID, entryTitle, entryArtist)
        return tplNewEntry


# 1. When program starts, read in the currently saved Inventory
FileProcessor.read_file(strFileName, lstTbl)

# 2. start main loop
while True:
    # 2.1 Display Menu to user and get choice
    IO.print_menu()
    strChoice = IO.menu_choice()
    # 3. Process menu selection
    # 3.1 process exit first
    if strChoice == 'x':
        break
    # 3.2 process load inventory
    if strChoice == 'l':
        print('WARNING: If you continue, all unsaved data will be lost and the Inventor
y re-loaded from file.')
```

```python
            strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled')
            if strYesNo.lower() == 'yes':
                print('reloading...')
                FileProcessor.read_file(strFileName, lstTbl)
                IO.show_inventory(lstTbl)
            else:
                input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
                IO.show_inventory(lstTbl)
            continue  # start loop back at top.
        # 3.3 process add a CD
        elif strChoice == 'a':
            # 3.3.0 get the sorted list of IDs currently in use
            lstTestIDs = DataProcessor.get_sorted_values_list(lstTbl)
            # 3.3.1 Ask user for new ID, CD Title and Artist
            tplUserEntry = IO.get_new_entry(lstTestIDs)
            # 3.3.2 Add item to the table
            # 3.3.2.1 Add item with users values to the table
            DataProcessor.add_CD(tplUserEntry, lstTbl)
            # 3.3.2.2 Display inventory to user to confirm CD added
            IO.show_inventory(lstTbl)
            continue  # start loop back at top.
        # 3.4 process display current inventory
        elif strChoice == 'i':
            IO.show_inventory(lstTbl)
            continue  # start loop back at top.
        # 3.5 process delete a CD
        elif strChoice == 'd':
            # 3.5.1 get Userinput for which CD to delete
            # 3.5.1.1 display Inventory to user
            IO.show_inventory(lstTbl)
            # 3.5.1.2 ask user which ID to remove
            intIDDel = int(input('Which ID would you like to delete? ').strip())
            # 3.5.2 search thru table and delete CD
            # 3.5.2.1 delete CD with requested ID
            DataProcessor.delete_CD(intIDDel, lstTbl)
            # 3.5.2.2 display inventory to user after delete for confirmation
            IO.show_inventory(lstTbl)
            continue  # start loop back at top.
        # 3.6 process save inventory to file
        elif strChoice == 's':
            # 3.6.1 Display current inventory and ask user for confirmation to save
            IO.show_inventory(lstTbl)
            strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
            # 3.6.2 Process choice
            if strYesNo == 'y':
                # 3.6.2.1 save data
                FileProcessor.write_file(strFileName, lstTbl)
            else:
                input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
            continue  # start loop back at top.
        # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be save:
        else:
            print('General Error')
```