

CD Inventory: Object Oriented Programming

Introduction

In this assignment I will explain the steps I took to modify a pre existing solution to the CD inventory program to incorporate object oriented programming. In addition to the new material, the script includes many elements to assignments 6 and 7 that dealt with classes, functions, and writing data to a text document.¹

Modifying the script

Data: CD Class

The first block in the SoC is the data block. I did not change anything here. This block established the main data variables for the script.

The first class defined in the data block is “class CD.” A class can be thought of as the blueprint for an object. The object that will be implemented in the code is an instantiation of the class: Each instance has the same functionality as the class, but is customized by its attributes.² In this case, only the docstring was provided for the CD class, so I had to add the code.

First I defined the constructor, which is “a dedicated method invoked when creating an object.”³ Constructors allow for default values, or pre-population. In this case, the constructor is going to make sure that the proper data fields are populated for our purposes. Each variable is defined in the attributes: cd_id, cd_title, cd_artist. Cd_id is cast as an integer, and since we are casting one of the values here, we have added error handling to catch if a user inputs a string instead of an integer for the cd_id.

¹ I have tried to avoid repeating information from the previous two assignments, and have thus not explained elif statements or Github in detail, or other components of the script that have remained the same from previous assignments.

² Mod 08 pg 2.

³ Mod 08 pg 3.

```

11 # -- DATA -- #
12 strChoice = ''
13 strFileName = 'cdInventory.txt'
14
15 class CD:
16     """Stores data about a CD:
17
18     properties:
19         cd_id: (int) with CD ID
20         cd_title: (string) with the title of the CD
21         cd_artist: (string) with the artist of the CD
22     methods:
23         append cd data to list of objects #need to format differently?
24
25     """
26     # TODO/done Add Code to the CD class
27
28     # -- Constructor -- #
29     def __init__(self, cd_id, cd_title, cd_artist):
30         # -- Attributes -- #
31         try: #Because we're casting, need error handling
32             self.__cd_id = int(cd_id)
33             self.__cd_title = cd_title
34             self.__cd_artist = cd_artist
35         except Exception as e:
36             raise Exception('Error setting initial values:\n' + str(e))
37

```

Figure 1: CD class with constructor and attributes.

Next come the properties. Since attributes are simply variables, the programmer has no control over them unless they write specific code to validate these values. The way to control these values is through properties, which have two parts per attribute: the getter and the setter. The getter retrieves/reads the attribute, and the setter runs/writes the attribute. The getter is always preceded by the `@property` decorator, and the setter preceded by the `@message.setter` decorator. These decorators are important because they denote separate functions -- even for functions with the same name. Because of the logic of how getters/setters work, and because getters can have the same name, getters must always be defined before setters.⁴

In this program, each getter refers to an attribute and references the object, which is called as `self`. Although `self` is not an official keyword in Python, it is commonly used and expected by all programmers.⁵ Each setter references the object with `self`, accepts values, and assigns those values to the classes private attributes. We know that the attributes are private because they are formatted with a dunder score.⁶

```

38 # -- Properties -- #
39
40 @property
41 def cd_id(self):
42     return self.__cd_id
43
44 @cd_id.setter
45 def cd_id(self, value):
46     self.__cd_id = value
47
48 @property
49 def cd_title(self):
50     return self.__cd_title
51
52 @cd_title.setter
53 def cd_title(self, value):
54     self.__cd_title = str(value)
55
56 @property
57 def cd_artist(self):
58     return self.__cd_artist
59
60 @cd_artist.setter
61 def cd_artist(self, value):
62     self.__cd_artist = str(value)
63

```

Figure 2: Properties.

⁴ Mod 08 pgs 6-7.

⁵ Mod 08 pg 5.

⁶ Mod 08 pg 7.

The final piece of the CD class are the methods. Methods are like functions within a class and can be called in the code, and methods reference the objects they are defined under -- that is why methods are assigned the *self* reference first.⁷ In this case we have two methods. First, the `print_neat` method, which, when called, will look at the CD object, pick out each CD individually, and print out a formatted string with info about each CD. The next method, `print_file`, is similar, but this function is specifically for writing the CD object to a file so that it is formatted in a way that is best for a text file.

```
# -- Methods -- #
def print_neat(self): # Looks at the CD object we called on - looks at one CD
    #and uses this function to print out a formatted string with info about this one CD.
    return '{}\t{} (by:{}'.format(self.cd_id, self.cd_title, self.cd_artist)

def print_file(self): # function for writing to file, CD object will now be formatted how
    return '{},{},{ }\n'.format(self.cd_id, self.cd_title, self.cd_artist)
```

Figure 3: Two methods in CD class.

Processing: FileIO class

The next SoC block is the Processing block. The first class here is FileIO. This class will process data to and from a text file by creating a function to load inventory from a text file, and also save inventory to a text file. The important things to note here are the following:

In the `load_inventory` function, we had to create an empty table to store the objects. In the error handling that follows, we make sure the file exists. If the file exists, each individual data point in the CD object is separated with a comma. Then, each row is ingested. The bracketed numbers refer to the individual data points of CD ID, title, and artist, in that order. If the file does not exist, it raises a `FileNotFoundError`, but the program can continue on without crashing.

In the `save_inventory` function, this function goes through each individual reference in the list of references to CD objects (`lst_inventory`). Each row is a reference to a CD object, which is then printed using the `print_file` method described above.

⁷ Mod 08 pg 10.

```

85 # TODO/done Add code to process data from a file
86 @staticmethod
87 def load_inventory(file_name): #can't reuse what we did in mod 7 because there we were p:
88     """ Function to load data from file to a list of CD objects
89
90     Reads the data from a file identified by file_name into a 2D table
91
92     Args:
93         file_name(string): name of file used to read data from
94
95     Returns:
96         2D list: list of CD Objects.
97     """
98     table = [] #table as list
99     try: # error handling to make sure file exists
100         with open(file_name, 'r') as f:
101             for line in f:
102                 data = line.strip().split(',')
103                 row = CD(data[0], data[1], data[2]) #each row ingested, each index refer:
104                 table.append(row)
105             return table
106
107     except FileNotFoundError as e: #trying to open a file that doesn't exist, or if they
108         print('Error: File {} could not be loaded'.format(file_name))
109         print('Error info:')
110         print(type(e), e, sep='\n')
111
112 # TODO Add code to process data to a file
113 @staticmethod
114 def save_inventory(file_name, lst_Inventory):
115     """ Function to save CD inventory (list of CD Objects) to file
116
117     Saves data from table to file with each on its own line, values are comma separated
118
119     Args:
120         file_name: name of file used to write the data to
121         lst_Inventory: list of CD Objects
122     Returns:
123         none
124     """
125     objFile = open(file_name, 'w')
126     for row in lst_Inventory: # list of references to CD objects, goes through each indiv
127         objFile.write(row.print_file()) # each row is a reference to a CD object. This e
128     objFile.close()
129

```

Figure 4: load_ and save_inventory functions.

Presentation: IO class

The next block in the SoC is the Presentation block and contains the class IO, which organizes functions that handle input and output. The first function in this class is print_menu(). The main thing to know here is that the delete function was removed for the purposes of this assignment. The next function here is menu_choice(); in this function, delete was also removed. These two functions remained otherwise unchanged from the previous assignments.

The next function under the IO class is show_inventory(table). This function contains a series of print statements that show the user what CDs are currently in memory. Here the print_neat() method from the CD class comes in handy to cycle through each CD object and show the individual values of each object.

```

# TODO add code to display the current data on screen
@staticmethod
def show_inventory(table):
    """Displays current inventory table

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

    Returns:
        None.

    """
    print('===== The Current Inventory: =====')
    print('ID\tCD Title (by: Artist)\n')
    for row in table:
        print(row.print_neat()) #cycling through each CD object and print_neat shows the individual values of
    print('=====')

```

Figure 5: show_inventory(table) utilizing the print_neat() method.

The final function in this class is get_user_input(), which remained unchanged from previous assignments.

Presentation: DataProcessor class

The final class in the Presentation SoC block is the DataProcessor class, which contains one function: `add_cd(CDInfo, table)`. This function takes the user input just described, creates a CD object, and then appends that object to a table, which holds a list of objects. Since there is casting involved in this function with the CD ID, this function also includes error handling to make sure that the user entered an integer for the CD ID.

```
class DataProcessor:
    """ Appends data to list of objects """

    @staticmethod
    def add_cd(CDInfo, table):
        """ Function to add CD to the inventory

        Args:
            CDInfo (tuple): Holds information to be added to inventory
        Returns:
            table (list of CDObjects): 2D data structure, list of dicts,

        """
        cdId, title, artist = CDInfo
        try:
            cdId = int(cdId)
        except ValueError as e: # error handling to make sure ID is an int
            print('ID is not an integer!')
            print(e.__doc__)
        row = CD(cdId, title, artist) #creates a CD object
        table.append(row) # appends the object to a table
        return table
```

Figure 6: DataProcessor class.

Main Body: What's New?

After calling the `load_inventory` function to load data from a text file onto the screen, the rest of the script is a series of `elif` statements nested under a `while` loop. Everything in this section had to be slightly revised to call the new functions described above. Once again, the `delete` `elif` statement was removed, because it was not needed for this version of the program.

Saving the file

To finish the file, I marked the TODOs as Done, removed some of the pseudocode to clean the script, and added my contributions to the header. I saved this file to a folder created for this course and renamed the file `CD_Inventory.py`.

Running the script, verifying functioning

To test the script, I first ran and tested all functions of the script in Spyder. Everything worked.

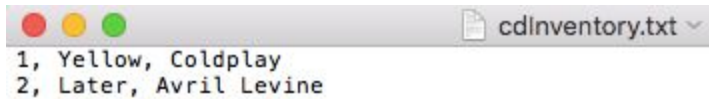
```
In [147]: runfile('/Users/Makena/Desktop/
Mod_08/CD_Inventory.py', wdir='/Users/
Makena/Desktop/Mod_08')
Error: File cdInventory.txt could not be
loaded
Error info:
<class 'FileNotFoundError'>
[Errno 2] No such file or directory:
'cdInventory.txt'
Menu

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

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

Figure 7: Working script in Spyder.

I checked the text file to make sure the data had been written in properly; it had.



```
cdInventory.txt
1, Yellow, Coldplay
2, Later, Avril Levine
```

Figure 8: cdInventory.txt with the written data.

I then tested the script from the terminal window, with different data inputs, from my desktop. It also worked properly. The new data was also in the text file.

```
[(base) makenas-air:~ Makena$ cd Desktop
[(base) makenas-air:Desktop Makena$ cd Mod_08
[(base) makenas-air:Mod_08 Makena$ python CD_Inventory.py
Error: File cdInventory.txt could not be loaded
Error info:
<class 'FileNotFoundError'>
[Errno 2] No such file or directory: 'cdInventory.txt'
Menu

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

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

Figure 16: The working script in IDLE.

Uploading to Github

Part of this assignment also involved saving the script to Github. To save this script (and this knowledge document) to Github, I created a repository called Assignment_08 and placed the files there. I then shared the link to this repository on the discussion board for this part of the assignment on canvas. Here is the link to my repository: <https://github.com/makenaflory/Assignment-08>

Summary

In this assignment I modified a pre existing solution to the CD inventory program to incorporate object oriented programming. In addition to the new material, the script includes many elements to assignments 6 and 7 that dealt with classes, functions, and writing data to a text document. The program allowed the user to view current inventory from in memory, load data from a text file, add new data to memory, and save data to a text file. This script was saved to Github.