Introduction to Structured Error Handling and the use of the Pickle function

# Introduction

This module focused on learning and applying structured error handling to allow the program created to better handle cases where errors occur without crashing the program.

# What is Structured Error Handling?

Structured error handling as a method used in programming languages to gracefully handle errors encountered in a code during run time. Python has a robust error handling framework, for which the RollBar Blog created a great article called “[Throwing Exceptions in Python](https://rollbar.com/blog/throwing-exceptions-in-python/),” which discusses the basics of error handling, how it’s implemented, and the logic behind it.

Python has an “Exception” class that has pre-defined values to handle a variety of errors that occur in run time. Additionally, Python several features that allow a user to define program specific errors which include the ‘raise’ and ‘assertation’ method. These methods allow the user to set out specific conditions that if they occur, this new error is generated.

When one of these pre-defined or user-defined errors is encountered, Python has a specific construct called ‘Try-Except’ which allows the programmer to account for the errors encountered and create alternative code paths for the computer to execute when an exception is thrown.

To use ‘try-except’ the user puts all code to be used in the ‘try’ section. If there are any errors encountered in this segment, these get kicked to the ‘except’ section for the cases created by the programmer. The code in the except section the runs to handle the exception case. Multiple exceptions can be created and used for each ‘try’ statement. Two additional statements can be used after, one is ‘else’, which only runs if the ‘try’ code runs without encountering any errors. The other is ‘finally’ which contains code that runs regardless the path the encountered.

# What is Pickling?

Pickling is a function in python that allows data in a program to be converted to and from a byte-stream format. This is a useful function because while humans are easily able to read text files, this is a much more cumbersome and memory intensive process for computers. Therefore, when text data is converted into a byte stream, it becomes much easer for the computer to process, which reduces computing and data transmission time. Real Python has an article ‘[The Python pickle Module: How to persist Objects in Python](https://realpython.com/python-pickle-module/),’ which outlines how the pickle module works and compares it to other methods as well.

The ‘pickle’ module has functionality to allow users to convert data to and from a byte stream within the program, or from a specified file.

# Applying Structured Error Handling and Pickling

These concepts were applied to the program created in the last module, which was adapted for these updates.

The first modification I did was to update functions in the ‘File Processor’ class to use the pickling method vice the read-write method I had previously been using for this class. This greatly simplified the code in these functions, since the pickle module supports any object type, so the data did not have to be converted to a string to perform this function. The new lines of code can be seen in the appendix from lines 118 – 141 and 144 – 158. There were only minor adaptations to the other areas of the code that had to be performed in order to accommodate the changes in the data structure that was loaded.

The rest of the modifications I performed were to assist with error handling and creating a more seamless product to run. First, I created ‘try-except’ structures around all places where a CD ID was input to ensure numerical data was being given to the program vice string.

I handled these instances with the ‘ValueError’ class of exception, which I implemented around the user input creating a loop to ensure the data was an integer before the loop could be exited. The code for this is on lines 102 – 110, 228 – 235, and 252 – 260.

The other built-in class of error that I handled with the ‘try-except’ structure was the ‘FileNotFoundError’ exception. This error was only encountered upon the first run of the program. This was a tricky error to handle, as not only was there no file, but there was no existing data structure for the rest of the functions to read from. I handled this with the ‘exception’ string of the program by having the branch open a write document, which automatically generates a document if one does not already exist. Then I created a dictionary with the same ‘key:value’ pairs that the program expected to see, but instead of normal values, I put in ‘None’ data types for the values. Doing this allowed for me to program in conditions in each function to read the values in the first line of the program to see if they were ‘None’ valued, and if so, the program would execute a different logic string so errors would not pop up. The main code adaptation is in the read\_file function, lines 118 – 141.

The other major bug that occurred when running this program was if all the CDs were deleted from the inventory, it caused the structure for the table of dictionaries to be erased. To handle this, I simply put in a clause in the delete\_CD function that said if the total number of rows in the table was 1 when deleting the last file, the a new ‘dummy row’ with ‘None’ type values as place holders for the keys is appended to the table, and then the code handled this new table the same way it handled the file creation case.

The last few pieces of code I created were simple functions to sort the data after CD deletion and addition, to ensure when new CDs with lower ID numbers were added, they were put to the top of the list (lines 72 – 82). Additionally, I created a bit of code to compare any CD ID with those already in the inventory to ensure no duplicate IDs were input into the system (lines 85 – 112).

# Running Script in Spyder IDE

The figures below show the result of running the ‘CDinventory.py’ script in Spyder IDE.

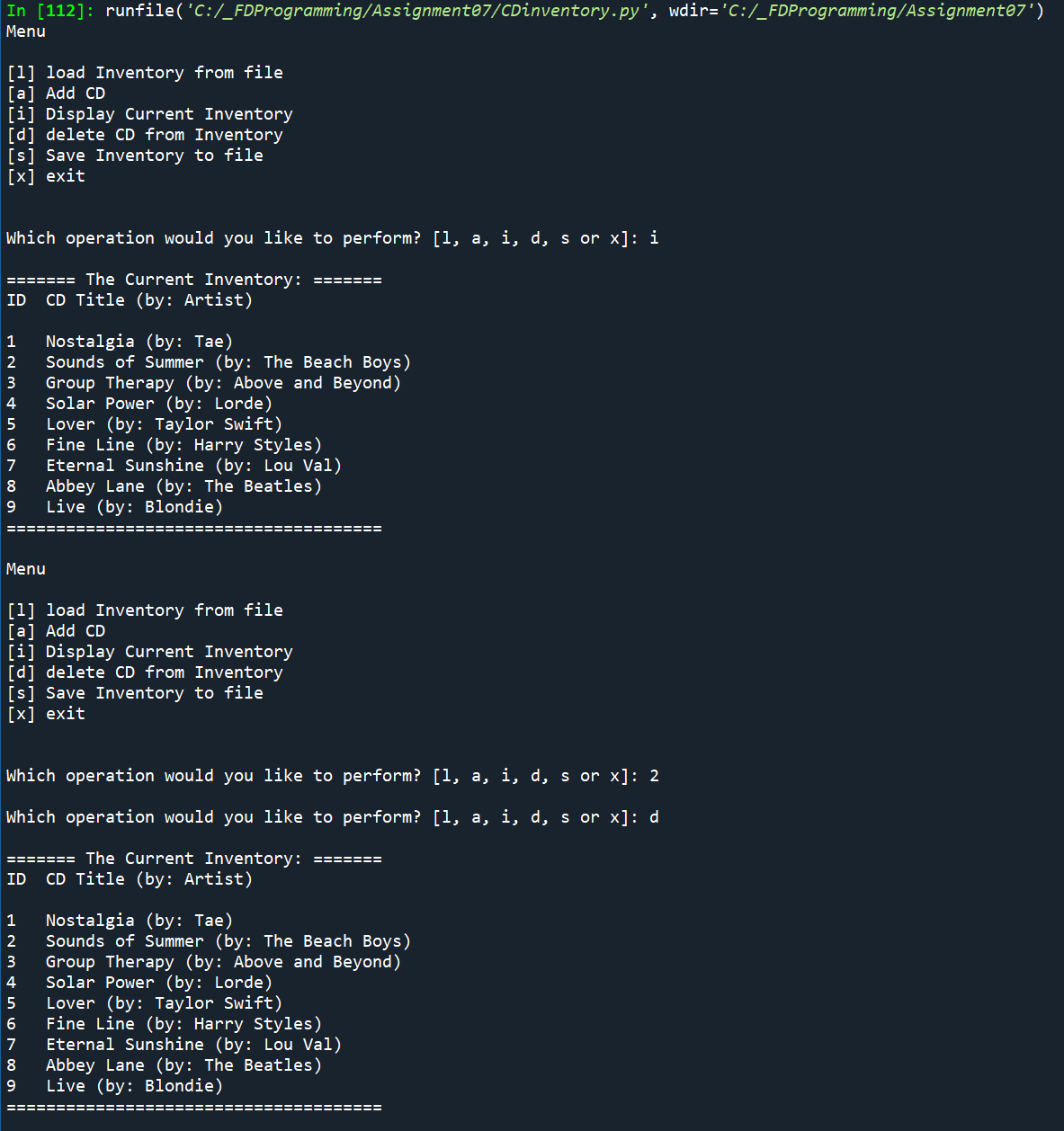


Figure - CDinventory.py program in Spyder

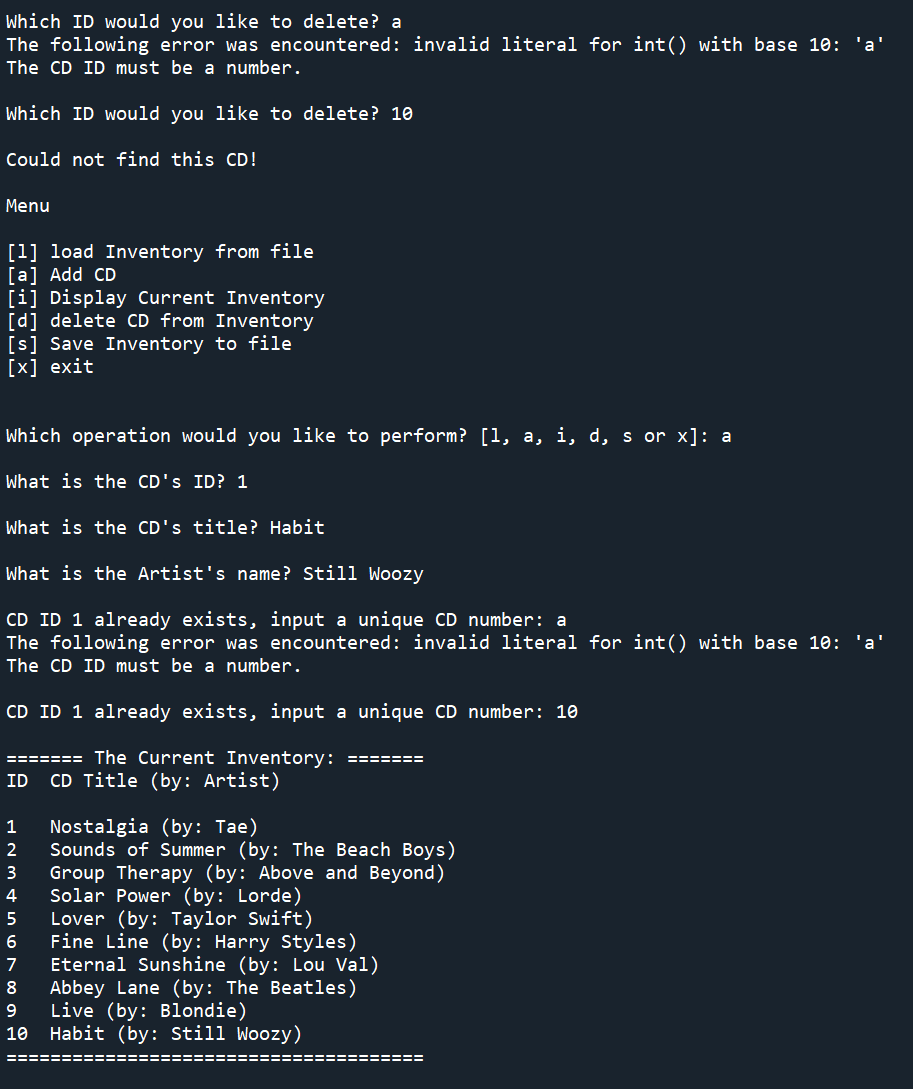


Figure 2 - CDInventory.py in Spyder Cont.

# Running Script in Terminal Window

The following figures are the result of running ‘CDInventory.py’ script in the terminal window.



Figure 4 - CDInventory.py Program run in Terminal Window

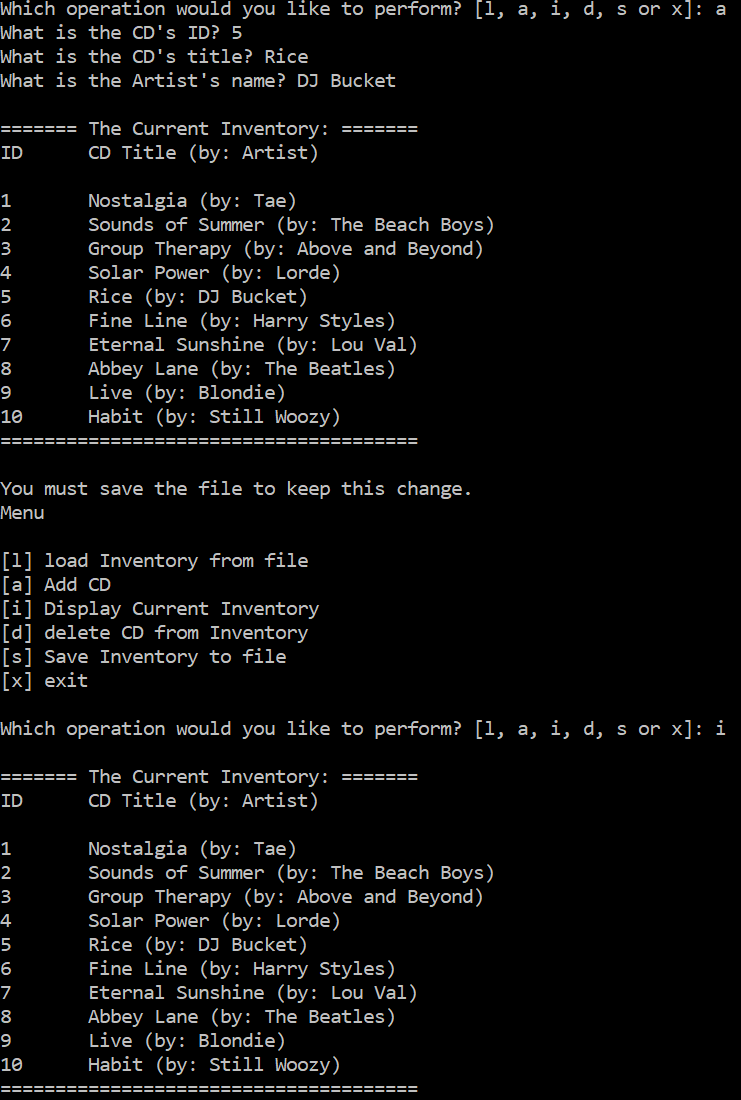


Figure 5 - CDInventory.py in Terminal Window Cont.

# Github Link

Here is the Github link to my code:

<https://github.com/mxlarkin/Assignment07/>

# Summary

In this assignment I learned how to implement error handling to reduce the amount of times the code I had written crashed. This is an important thing to do when creating a code, however it can be difficult to determine where to best place ‘try-except’ statements to appropriately handle errors where they are most likely to occur. It is also difficult to determine what kind of code to put in the ‘except’ statement that will not cause the code to continue running without all the inputs necessary to appropriately function.

I also learned how to use the pickle module, which is a much simple application than the previous use of read and write, as the data structures being pickled can be any type of object, and not just a string, this greatly simplifies the amount of data processing that needs to occur to get data into and out of the program.

# Appendix

CDinventory Python script:

1. *#------------------------------------------#*
2. *# Title: CDinventory.py*
3. *# Desc: Created a file to*
4. *# Change Log: (Who, When, What)*
5. ***# DBiesinger, 2030-Jan-01, Created File***
6. *# Larkin, 2021-Aug-15, Edited File, converted to structure format*
7. *# Larkin, 2021-Aug-23, Edited File, added Error handling, pickling*
8. *#------------------------------------------#*
10. ***# -- DATA -- #***
11. **import** pickle
13. strChoice = '' *# User input*
14. dicTbl = [] *# list of dicationaries to hold data*
15. **dicRow = {} *# dictionary of data row***
16. strFileName = 'CDInventory.dat' *# data storage file*
17. objFile = None *# file object*

20. ***# -- PROCESSING -- #***
21. **class** DataProcessor:
22. '''Adding and deleting items from the CD inventory table stored during run time'''
23. *# TODone add functions for processing here*
25. **@staticmethod**
26. **def** add\_CD(cdID, title, artist, table):
27. '''Function to add CD to 2D table, list of dics
28. Args:
29. cdID (string): string to add to dictionary
30. **title (string): string to add to dictionary**
31. artist (string): string to add to dictionary
32. table (list of Dics): Table of dics to add new dictionary to
33. Returns:
34. table (list of Dics): Table with added CD
35. **'''**
37. dicRow = {'ID': cdID, 'Title': title, 'Artist': artist}
38. **if** table[0]['ID'] == None:
39. table = [dicRow]
40. **else:**
41. table.append(dicRow)
42. **return** table
44. @staticmethod
45. **def delete\_CD(cdID, table):**
46. '''Function to delete CD in ED table, list of dics
47. Args:
48. cdID (Integer): ID number for CD in dictionary to delete from table
49. table (list of dics): Table of dictionaries to delete dic from
50. **Returns:**
51. blnCDRemoved (boolean): Boolean to allow program to know if CD was deleted
52. table (list of dics): updated table after CD deletion
53. '''
54. intRowNr = -1
55. **blnCDRemoved = False**
56. **if** table[0]['ID'] == None:
57. blnCDRemoved = False
58. **else**:
60. **for row in table:**
61. intRowNr += 1
62. **if** int(row['ID']) == cdID:
63. **if** len(table) == 1:
64. data = {'ID': None, 'Title': None, 'Artist': None}
65. **table.append(data)**
66. **del** table[intRowNr]
67. blnCDRemoved = True
68. **break**
69. **return** blnCDRemoved, table
71. @staticmethod
72. **def** sort\_CD(table):
73. '''Function to sort list of dictionarires
74. Args:
75. **table (list of dictionaries): List of CDs to be sorted**
76. Returns:
77. sorted\_table (list of dictionaries): sorted list of CDs'''
78. **if** table[0]['ID'] == None:
79. sorted\_table = table
80. **else:**
81. sorted\_table = sorted(table, key = **lambda** i: i['ID'])
82. **return** sorted\_table
84. @staticmethod
85. **def compare\_CD(cdID, table):**
86. '''Function to compare new CD ID with existing IDs
87. Args:
88. cdID (integer): ID of CD to be added
89. table (list of dics): Table of existing CDs
90. **Returns:**
91. cdID (integer): returns CD ID once it is unique
92. '''
93. **print**(table[0]['ID'])
94. **if** table[0]['ID'] == None:
95. **print('There are no CDs to compare.')**
96. **else**:
97. boolean = True
98. **while** boolean:
99. **for** i **in** range(len(table)):
100. **j = table[i]['ID']**
101. **if** j == cdID:
102. boolean2 = True
103. **while** boolean2:
104. **try**:
105. **cdID = int(input('CD ID ' + str(cdID) + ' already exists, input a unique CD number: '))**
106. boolean2 = False
107. **except** ValueError **as** e:
108. **print**('The following error was encountered:', e)
109. **print**('The CD ID must be a number.')
110. **elif i == len(table) - 1:**
111. boolean = False
112. **return** cdID
114. **class** FileProcessor:
115. **"""Processing the data to and from text file"""**
117. @staticmethod
118. **def** read\_file(file\_name, table):
119. """Function to retrieve and unserialize data from saved file
121. Accesses and unserializes data from a data file
123. Args:
124. file\_name (string): name of file used to get data from
125. **table (list of dict): 2D data structure (list of dicts) that holds the data during runtime**
127. Returns:
128. dataTbl (list of dictionaries): Holds CD file inputs
129. """
130. **table.clear() *# this clears existing data and allows to load data from file***
131. **try**:
132. **with** open(file\_name, 'rb') **as** file:
133. dataTbl = pickle.load(file)
134. **except** FileNotFoundError **as** e:
135. **print('The following error was encountered:', e)**
136. **with** open(file\_name, 'wb') **as** file:
137. **print**(file\_name, 'has been created!')
138. data = [{'ID': None, 'Title': None, 'Artist': None}]
139. pickle.dump(data, file)
140. **dataTbl = data**
141. **return** dataTbl
143. @staticmethod
144. **def** write\_file(file\_name, table):
145. **'''Function to save serialized user data to a file using the pickle function**
147. Serializes object that is entered into table
149. Args:
150. **file\_name: Name of file to save serialized table to**
151. table: 2D data structure (list of dictionaries) that holds data during run time
152. Returns:
153. None.
154. '''
155. ***# TODone Add code here***
157. **with** open(file\_name, 'wb') **as** file:
158. pickle.dump(table, file)

161. *# -- PRESENTATION (Input/Output) -- #*
163. **class** IO:
164. """Handling Input / Output"""
166. @staticmethod
167. **def** print\_menu():
168. """Displays a menu of choices to the user
170. **Args:**
171. None.
173. Returns:
174. None.
175. **"""**
177. **print**('Menu**\n\n**[l] load Inventory from file**\n**[a] Add CD**\n**[i] Display Current Inventory')
178. **print**('[d] delete CD from Inventory**\n**[s] Save Inventory to file**\n**[x] exit**\n**')
180. **@staticmethod**
181. **def** menu\_choice():
182. """Gets user input for menu selection
184. Args:
185. **None.**
187. Returns:
188. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
189. """
190. **choice = ''**
191. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
192. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
193. *#print('\n') # Add extra space for layout*
194. **return** choice
196. @staticmethod
197. **def** show\_inventory(table):
198. """Displays current inventory table

201. Args:
202. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
204. Returns:
205. **None.**
207. """
208. **print**('**\n**======= The Current Inventory: =======')
209. **print**('ID**\t**CD Title (by: Artist)**\n**')
210. **if table[0]['ID'] == None:**
211. **print**()
212. **else**:
213. **for** row **in** table:
214. **print**('{}**\t**{} (by: {})'.format(\*row.values()))
215. **print('======================================\n')**
217. *# TODone add I/O functions as needed*
218. @staticmethod
219. **def** input\_CD():
220. **'''Get user input for CD to add**
221. Args:
222. None
223. Returns:
224. strID (integer): string of user input for new CD ID
225. **strTitle (string): string with user input for new CD title**
226. strArtist (string): string with user input for new CD artist
227. '''
228. boolean = True
229. **while** boolean:
230. **try:**
231. cdID = int(input('What is the CD**\'**s ID? ').strip())
232. boolean = False
233. **except** ValueError **as** e:
234. **print**('The following error was encountered:', e)
235. **print('The CD ID must be a number.')**
236. title = input('What is the CD**\'**s title? ').strip()
237. artist = input('What is the Artist**\'**s name? ').strip()
238. **return** cdID, title, artist
240. **@staticmethod**
241. **def** input\_ID(table):
242. '''Function to get CD ID for deleted CD
243. Args:
244. table (list of dics): to test if inital value is None
245. **Returns:**
246. intIDDel (integer): number of ID to delete
247. '''
248. **if** table[0]['ID'] == None:
249. **print**('There are no CDs to remove.**\n**')
250. **intIDDel = 0**
251. **else**:
252. boolean = True
253. **while** boolean:
254. **try**:
255. **intIDDel = int(input('Which ID would you like to delete? ').strip())**
256. boolean = False
257. **except** ValueError **as** e:
258. **print**('The following error was encountered:', e)
259. **print**('The CD ID must be a number.')
260. **return intIDDel**
262. @staticmethod
263. **def** cd\_status(boolean):
264. '''Function to determine if CD was removed from stack
265. **Args:**
266. boolean (bool): Determines if CD is still there or not
267. Returns:
268. None
269. '''
270. **if blnCDRmvd:**
271. **print**('**\n**The CD was removed.**\n**')
272. **else**:
273. **print**('**\n**Could not find this CD!**\n**')

276. *# 1. When program starts, read in the currently saved Inventory*
277. dicTbl = FileProcessor.read\_file(strFileName, dicTbl)
279. *# 2. start main loop*
280. **while True:**
281. *# 2.1 Display Menu to user and get choice*
282. IO.print\_menu()
283. strChoice = IO.menu\_choice()
285. ***# 3. Process menu selection***
286. *# 3.1 process exit first*
287. **if** strChoice.lower() == 'x':
288. **break**
289. *# 3.2 process load inventory*
290. **if strChoice.lower() == 'l':**
291. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
292. strYesNo = input('Type **\'**y**\'** to continue and reload from file; otherwise reload will be cancelled:**\n**')
293. **if** strYesNo.lower() == 'y':
294. **print**('reloading...')
295. **dicTbl = FileProcessor.read\_file(strFileName, dicTbl)**
296. IO.show\_inventory(dicTbl)
297. **else**:
298. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
299. IO.show\_inventory(dicTbl)
300. **continue *# start loop back at top.***
301. *# 3.3 process add a CD*
302. **elif** strChoice.lower() == 'a':
303. *# 3.3.1 Ask user for new ID, CD Title and Artist*
304. *# TODone move IO code into function*
305. **strID, strTitle, strArtist = IO.input\_CD()**
306. *# Compare supplied CD ID to existing CD IDs*
307. strID = DataProcessor.compare\_CD(strID, dicTbl)
308. *# 3.3.2 Add item to the table*
309. *# TODone move processing code into function*
310. **dicTbl = DataProcessor.add\_CD(strID, strTitle, strArtist, dicTbl)**
311. *# Sorts CD inventory*
312. dicTbl = DataProcessor.sort\_CD(dicTbl)
313. IO.show\_inventory(dicTbl)
314. **print**('You must save the file to keep this change.')
315. **continue *# start loop back at top.***
316. *# 3.4 process display current inventory*
317. **elif** strChoice.lower() == 'i':
318. IO.show\_inventory(dicTbl)
319. **continue** *# start loop back at top.*
320. ***# 3.5 process delete a CD***
321. **elif** strChoice.lower() == 'd':
322. *# 3.5.1 get Userinput for which CD to delete*
323. *# 3.5.1.1 display Inventory to user*
324. IO.show\_inventory(dicTbl)
325. ***# 3.5.1.2 ask user which ID to remove***
326. intIDDel = IO.input\_ID(dicTbl)
327. *# 3.5.2 search thru table and delete CD*
328. *# TODone move processing code into function*
329. blnCDRmvd, dicTbl = DataProcessor.delete\_CD(intIDDel, dicTbl)
330. ***# Tell user if CD was able to be removed or not***
331. **if** dicTbl[0]['ID'] != None:
332. IO.cd\_status(blnCDRmvd)
333. **if** blnCDRmvd:
334. dicTbl = DataProcessor.sort\_CD(dicTbl)
335. ***# Show the inventory with the CD removed***
336. IO.show\_inventory(dicTbl)
337. **print**('You must save the file to keep this change.')
338. **continue** *# start loop back at top.*
339. *# 3.6 process save inventory to file*
340. **elif strChoice.lower() == 's':**
341. *# 3.6.1 Display current inventory and ask user for confirmation to save*
342. IO.show\_inventory(dicTbl)
343. strYesNo = input('Type **\'**y**\'** to save data to file; otherwise save will be cancelled:**\n**').strip().lower()
344. *# 3.6.2 Process choice*
345. **if strYesNo.lower() == 'y':**
346. *# 3.6.2.1 save data*
347. *# TODone move processing code into function*
348. FileProcessor.write\_file(strFileName, dicTbl)
349. **else**:
350. **input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')**
351. **continue** *# start loop back at top.*
352. *# 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be save:*
353. **else**:
354. **print**('General Error')