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 form lines 101 – 121 and 124 – 138. 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 86 – 92, 208 – 212, and 225 – 232.

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. In order to combat this, I simply had the except clause create a file of the appropriate name if one did not already exist. The code for this is in lines 114 – 120.

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 61 – 69). 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 72 – 95).

# Running Script in Spyder IDE

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

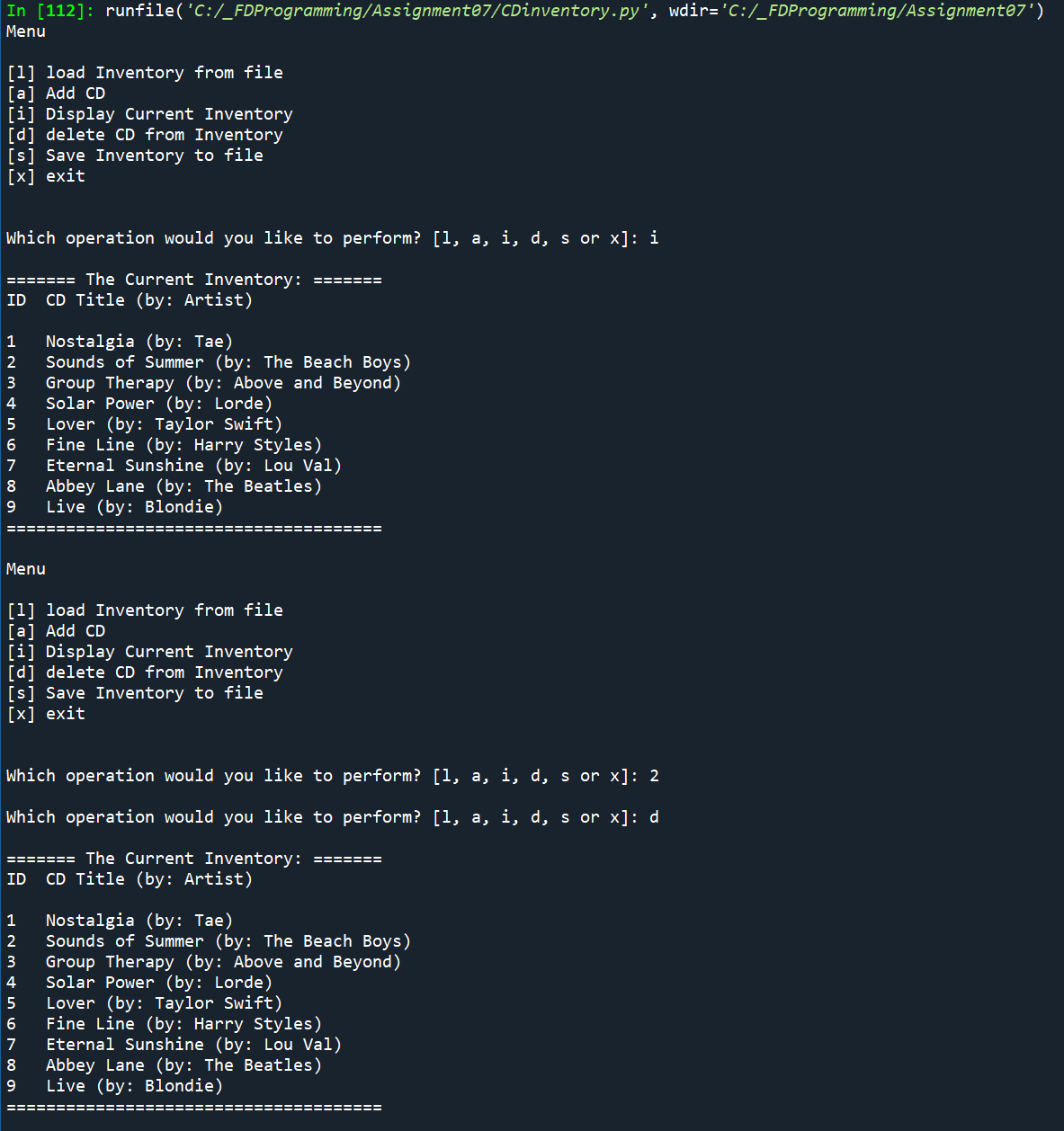


Figure 1 - CDinventory.py program in Spyder

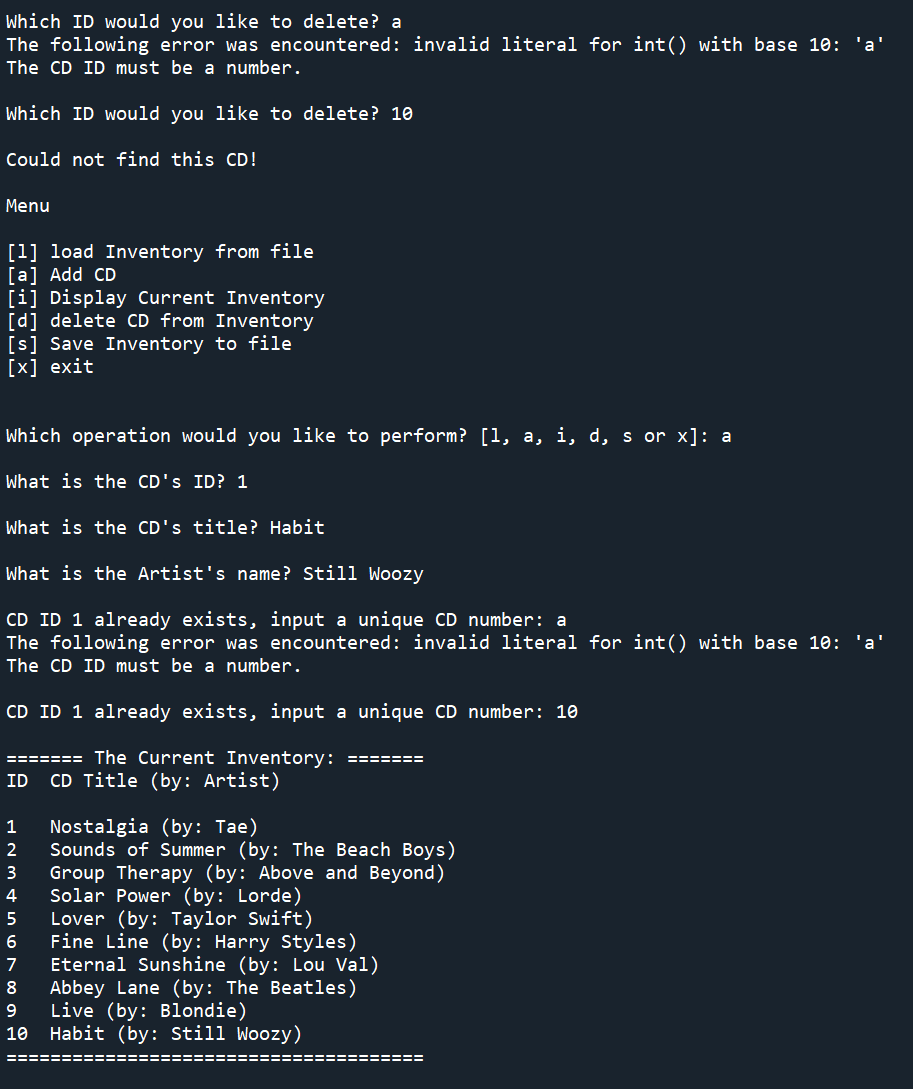


Figure - CDInventory.py in Spyder Cont.

# 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

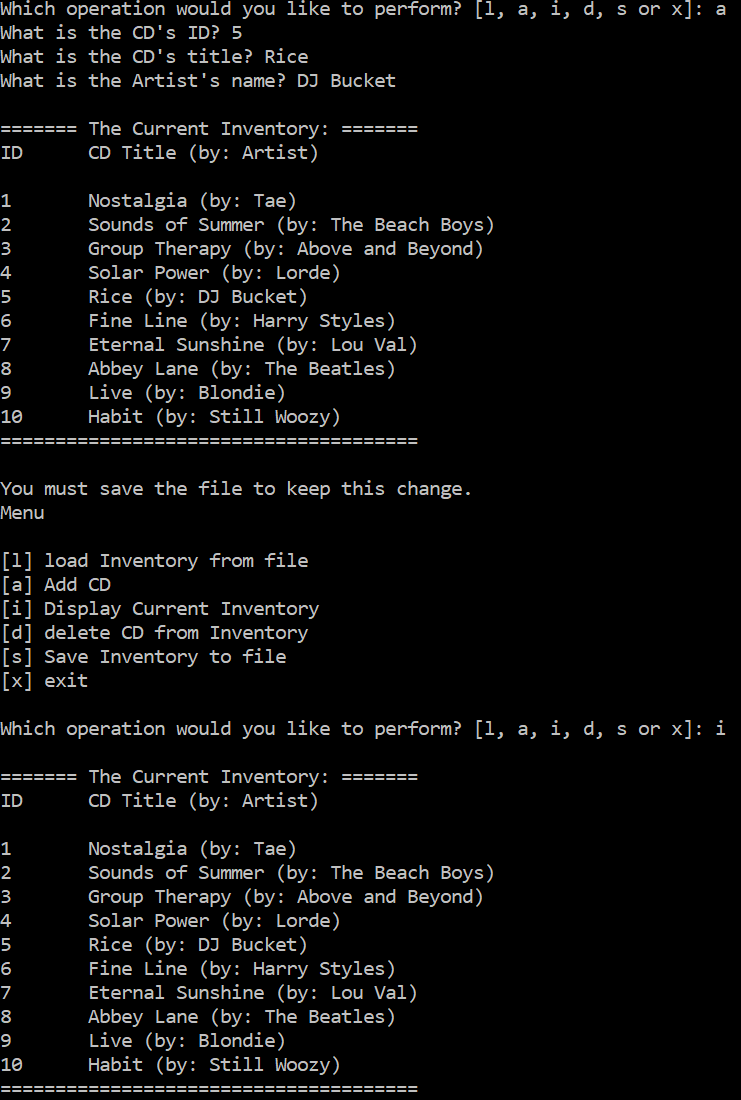


Figure 5 - CDInventory.py in Terminal Window Cont.

# 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. **'''**
36. dicRow = {'ID': cdID, 'Title': title, 'Artist': artist}
37. table.append(dicRow)
38. **return** table
40. **@staticmethod**
41. **def** delete\_CD(cdID, table):
42. '''Function to delete CD in ED table, list of dics
43. Args:
44. cdID (Integer): ID number for CD in dictionary to delete from table
45. **table (list of dics): Table of dictionaries to delete dic from**
46. Returns:
47. blnCDRemoved (boolean): Boolean to allow program to know if CD was deleted
48. table (list of dics): updated table after CD deletion
49. '''
50. **intRowNr = -1**
51. blnCDRemoved = False
52. **for** row **in** table:
53. intRowNr += 1
54. **if** int(row['ID']) == cdID:
55. **del table[intRowNr]**
56. blnCDRemoved = True
57. **break**
58. **return** blnCDRemoved, table
60. **@staticmethod**
61. **def** sort\_CD(table):
62. '''Function to sort list of dictionarires
63. Args:
64. table (list of dictionaries): List of CDs to be sorted
65. **Returns:**
66. sorted\_table (list of dictionaries): sorted list of CDs'''
68. sorted\_table = sorted(table, key = **lambda** i: i['ID'])
69. **return** sorted\_table
71. @staticmethod
72. **def** compare\_CD(cdID, table):
73. '''Function to compare new CD ID with existing IDs
74. Args:
75. **cdID (integer): ID of CD to be added**
76. table (list of dics): Table of existing CDs
77. Returns:
78. cdID (integer): returns CD ID once it is unique
79. '''
80. **boolean = True**
81. **while** boolean:
82. **for** i **in** range(len(table)):
83. j = table[i]['ID']
84. **if** j == cdID:
85. **boolean2 = True**
86. **while** boolean2:
87. **try**:
88. cdID = int(input('CD ID ' + str(cdID) + ' already exists, input a unique CD number: '))
89. boolean2 = False
90. **except ValueError as e:**
91. **print**('The following error was encountered:', e)
92. **print**('The CD ID must be a number.')
93. **elif** i == len(table) - 1:
94. boolean = False
95. **return cdID**
97. **class** FileProcessor:
98. """Processing the data to and from text file"""
100. **@staticmethod**
101. **def** read\_file(file\_name, table):
102. """Function to retrieve and unserialize data from saved file
104. Accesses and unserializes data from a data file
106. Args:
107. file\_name (string): name of file used to get data from
108. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
110. **Returns:**
111. dataTbl (list of dictionaries): Holds CD file inputs
112. """
113. table.clear() *# this clears existing data and allows to load data from file*
114. **try**:
115. **with open(file\_name, 'rb') as file:**
116. dataTbl = pickle.load(file)
117. **except** FileNotFoundError **as** e:
118. **print**('The following error was encountered:', e)
119. **with** open(file\_name, 'xb') **as** file:
120. **print(file\_name, 'has been created!')**
121. **return** dataTbl
123. @staticmethod
124. **def** write\_file(file\_name, table):
125. **'''Function to save serialized user data to a file using the pickle function**
127. Serializes object that is entered into table
129. Args:
130. **file\_name: Name of file to save serialized table to**
131. table: 2D data structure (list of dictionaries) that holds data during run time
132. Returns:
133. None.
134. '''
135. ***# TODone Add code here***
137. **with** open(file\_name, 'wb') **as** file:
138. pickle.dump(table, file)

141. *# -- PRESENTATION (Input/Output) -- #*
143. **class** IO:
144. """Handling Input / Output"""
146. @staticmethod
147. **def** print\_menu():
148. """Displays a menu of choices to the user
150. **Args:**
151. None.
153. Returns:
154. None.
155. **"""**
157. **print**('Menu**\n\n**[l] load Inventory from file**\n**[a] Add CD**\n**[i] Display Current Inventory')
158. **print**('[d] delete CD from Inventory**\n**[s] Save Inventory to file**\n**[x] exit**\n**')
160. **@staticmethod**
161. **def** menu\_choice():
162. """Gets user input for menu selection
164. Args:
165. **None.**
167. Returns:
168. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
169. """
170. **choice = ''**
171. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
172. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
173. *#print('\n') # Add extra space for layout*
174. **return** choice
176. @staticmethod
177. **def** show\_inventory(table):
178. """Displays current inventory table

181. Args:
182. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
184. Returns:
185. **None.**
187. """
188. **print**('**\n**======= The Current Inventory: =======')
189. **print**('ID**\t**CD Title (by: Artist)**\n**')
190. **for row in table:**
191. **print**('{}**\t**{} (by: {})'.format(\*row.values()))
192. **print**('======================================**\n**')
194. *# TODone add I/O functions as needed*
195. **@staticmethod**
196. **def** input\_CD():
197. '''Get user input for CD to add
198. Args:
199. None
200. **Returns:**
201. strID (integer): string of user input for new CD ID
202. strTitle (string): string with user input for new CD title
203. strArtist (string): string with user input for new CD artist
204. '''
205. **boolean = True**
206. **while** boolean:
207. **try**:
208. cdID = int(input('What is the CD**\'**s ID? ').strip())
209. boolean = False
210. **except ValueError as e:**
211. **print**('The following error was encountered:', e)
212. **print**('The CD ID must be a number.')
213. title = input('What is the CD**\'**s title? ').strip()
214. artist = input('What is the Artist**\'**s name? ').strip()
215. **return cdID, title, artist**
217. @staticmethod
218. **def** input\_ID():
219. '''Function to get CD ID for deleted CD
220. **Args:**
221. None
222. Returns:
223. intIDDel (integer): number of ID to delete
224. '''
225. **boolean = True**
226. **while** boolean:
228. **try**:
229. intIDDel = int(input('Which ID would you like to delete? ').strip())
230. **boolean = False**
231. **except** ValueError **as** e:
232. **print**('The following error was encountered:', e)
233. **print**('The CD ID must be a number.')
234. **return** intIDDel
236. @staticmethod
237. **def** cd\_status(boolean):
238. '''Function to determine if CD was removed from stack
239. Args:
240. **boolean (bool): Determines if CD is still there or not**
241. Returns:
242. None
243. '''
244. **if** blnCDRmvd:
245. **print('\nThe CD was removed.\n')**
246. **else**:
247. **print**('**\n**Could not find this CD!**\n**')

250. ***# 1. When program starts, read in the currently saved Inventory***
251. dicTbl = FileProcessor.read\_file(strFileName, dicTbl)
253. *# 2. start main loop*
254. **while** True:
255. ***# 2.1 Display Menu to user and get choice***
256. IO.print\_menu()
257. strChoice = IO.menu\_choice()
259. *# 3. Process menu selection*
260. ***# 3.1 process exit first***
261. **if** strChoice == 'x':
262. **break**
263. *# 3.2 process load inventory*
264. **if** strChoice == 'l':
265. **print('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')**
266. strYesNo = input('Type **\'**y**\'** to continue and reload from file; otherwise reload will be cancelled:**\n**')
267. **if** strYesNo.lower() == 'y':
268. **print**('reloading...')
269. dicTbl = FileProcessor.read\_file(strFileName, dicTbl)
270. **IO.show\_inventory(dicTbl)**
271. **else**:
272. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
273. IO.show\_inventory(dicTbl)
274. **continue** *# start loop back at top.*
275. ***# 3.3 process add a CD***
276. **elif** strChoice == 'a':
277. *# 3.3.1 Ask user for new ID, CD Title and Artist*
278. *# TODone move IO code into function*
279. strID, strTitle, strArtist = IO.input\_CD()
280. ***# Compare supplied CD ID to existing CD IDs***
281. strID = DataProcessor.compare\_CD(strID, dicTbl)
282. *# 3.3.2 Add item to the table*
283. *# TODone move processing code into function*
284. dicTbl = DataProcessor.add\_CD(strID, strTitle, strArtist, dicTbl)
286. dicTbl = DataProcessor.sort\_CD(dicTbl)
287. IO.show\_inventory(dicTbl)
288. **print**('You must save the file to keep this change.')
289. **continue** *# start loop back at top.*
290. ***# 3.4 process display current inventory***
291. **elif** strChoice == 'i':
292. IO.show\_inventory(dicTbl)
293. **continue** *# start loop back at top.*
294. *# 3.5 process delete a CD*
295. **elif strChoice == 'd':**
296. *# 3.5.1 get Userinput for which CD to delete*
297. *# 3.5.1.1 display Inventory to user*
298. IO.show\_inventory(dicTbl)
299. *# 3.5.1.2 ask user which ID to remove*
300. **intIDDel = IO.input\_ID()**
301. *# 3.5.2 search thru table and delete CD*
302. *# TODone move processing code into function*
303. blnCDRmvd, dicTbl = DataProcessor.delete\_CD(intIDDel, dicTbl)
304. *# Tell user if CD was able to be removed or not*
305. **IO.cd\_status(blnCDRmvd)**
306. **if** blnCDRmvd:
307. dicTbl = DataProcessor.sort\_CD(dicTbl)
308. *# Show the inventory with the CD removed*
309. IO.show\_inventory(dicTbl)
310. **print('You must save the file to keep this change.')**
311. **continue** *# start loop back at top.*
312. *# 3.6 process save inventory to file*
313. **elif** strChoice == 's':
314. *# 3.6.1 Display current inventory and ask user for confirmation to save*
315. **IO.show\_inventory(dicTbl)**
316. strYesNo = input('Type **\'**y**\'** to save data to file; otherwise save will be cancelled:**\n**').strip().lower()
317. *# 3.6.2 Process choice*
318. **if** strYesNo == 'y':
319. *# 3.6.2.1 save data*
320. ***# TODone move processing code into function***
321. FileProcessor.write\_file(strFileName, dicTbl)
322. **else**:
323. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
324. **continue** *# start loop back at top.*
325. ***# 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be save:***
326. **else**:
327. **print**('General Error')