Christopher J. Scott

1-Mar-2021

IT FDN 110: Introduction to Programming (Python)

Assignment 07

Working with binary data files and structured error handling

# Introduction

Module 07 expanded on working with text files, generating and using binary files, structured error handling and more advances github utilization. These new lessons were incorporated into Assignment 07 which added on the the program generated in Assignment 06.

# Binary data files and ways to manipulate them

Module 7 presents multiple ways to interact with files including the previously introduced write(), read(), append() functions which serve to “save to”, “open from” and “add to” respectively. Other options for reading data include readline() and readlines(). The readline() function returns the data within the “next” line in a data set which is referred to as a cursor while the readlines() function reads all the lines in a file and returns a list.

The benefits of utilizing binary files include improved efficiency in handling data when it is not critical for it to be legible for a human being. There is no effort required for specific formatting and with larger data files, this can be a real advantage.

NOTE: it is important to remember that serializing data, or saving as binary, is NOT encrypting and should not be used as a means to protect sensitive information.

The module that existing within Python for working with binary data is pickle(). This pickle module has the ability to take information associated with an object and serialize, or pickle, it. The pickled data can be used within a Python program if it is not imperative for the data to be formatted for human consumption.



Figure 1. pickle.load and pickle.dump being called upon in Assignment07

It’s important to note that the argument ‘rb’ and ‘wb are critical for reading (r) to a file using binary (b) mode and writing (w) using binary (b) mode respectively. The following website goes into greater detail in regards to object serialization and the pickle module: <https://www.datacamp.com/community/tutorials/pickle-python-tutorial>.

# Structured error handling

Throughout this class, we have inadvertently encountered many types of errors when a user would input data or interact with the program in a way that was not expected or intended. Accounting for these instances is critical in ensuring that a program will continue to execute when it encounters these situations. In order to reduce the risk of instances where a program will crash or generate an error that disrupts the script, a programmer may choose to invke error handling. This can be done by attempting to trap the problematic statements and put in indicators that will alert the user and potentially handle the error by executing a specific portion of code.

Some examples of errors that we have seen through out the course include ValueError, TypeError and ZeroDivisionError. These are examples where the program may have expected a certain type of input or a certain value and these errors were generated when the program could not compute.

When working to trap specific errors, the programmer may include multiple specific exceptions that will execute when a particular error type happens. Figure 2 (Listing 14 in Module 07) shows a few different error types and the way that a programmer can pass information about that error to the user using print() commands and also ways to dig into details contained in the class of the error.

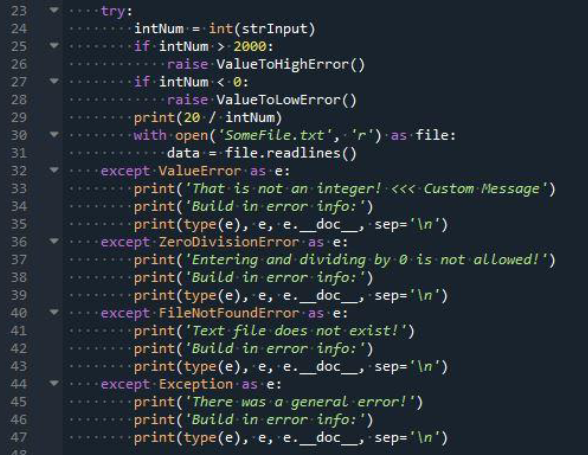


Figure 2. try-except demonstrated in Module 07

# Putting it all together in Assignment 07

In order to apply the knowledge learned in Module 07, we first started with the code that was completed in Assignment 06. I took the code that I created and first incorporated the comments from Doug in order to improve my code and ensure that it performed as intended.

The instances where I incorporated error handling are anywhere there was user input/output into the program, type casting (string type to int type) and accessing files. Some examples of that can be seen below in Figure 3 and Figure 4.

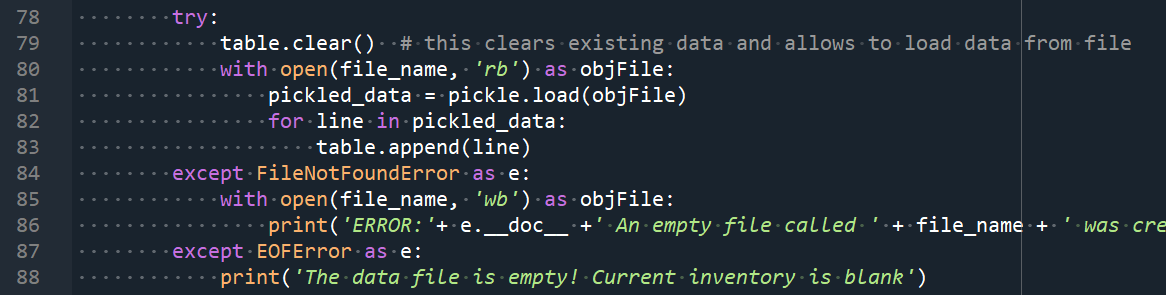


Figure 3. Error handling around data loading

When opening the .dat file, two error handling situations were introduced to catch scenarios where a .dat file does not exist or it contains empty information.

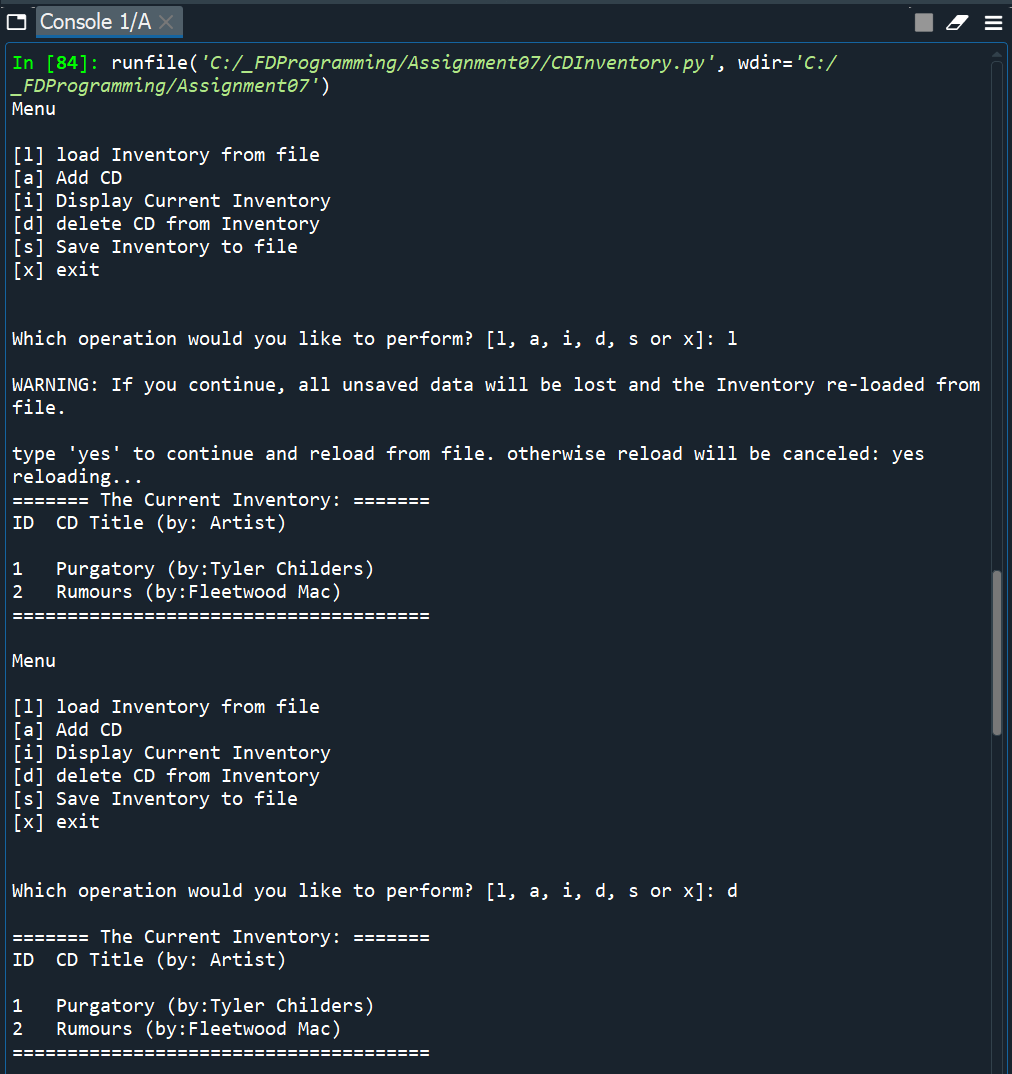
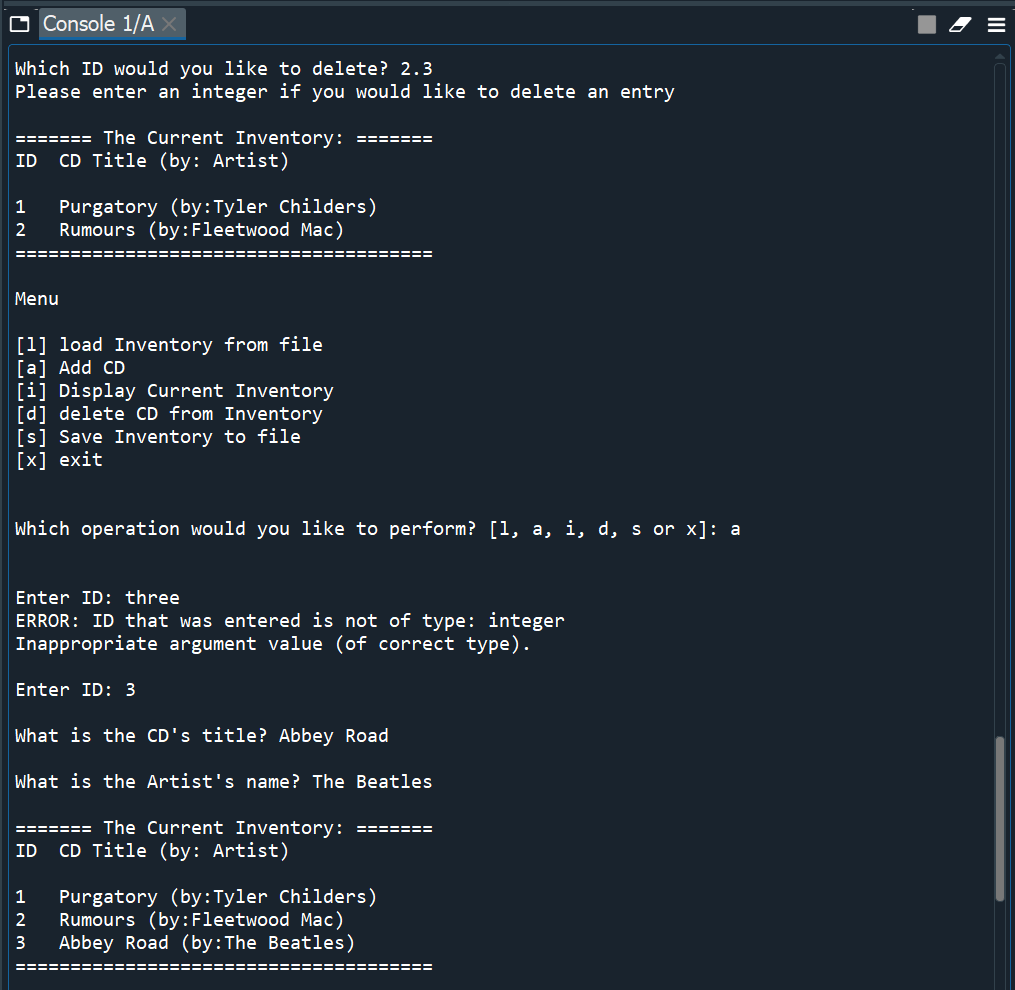
 

Figure 4. Demonstrating error handling around TypeError and ValueError

Upon saving the inventory to a binary file using pickle.dump(), the CDInventory.dat file results in Figure 5 which can be seen to not make much sense to a human reading the file.

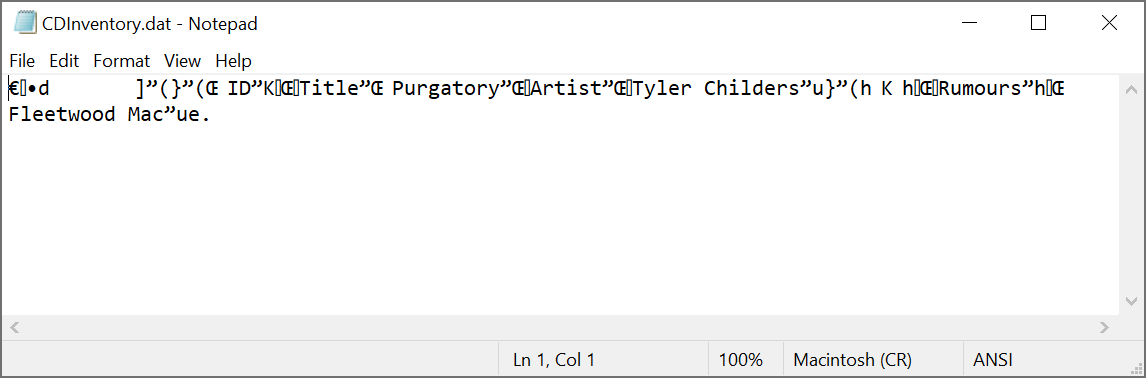


Figure 5. Result of saving within CDInventory.py

# Summary

By completing Module 7 and associated Lab exercises, a user is able to create a program that utilizes binary data files and error handling to minimize the chance of a user breaking a code and having it crash. As was the case in previous modules, a repository that contains this specific CDInventory.py for Assignment 97 can be found at <https://github.com/cscott08vt/Assignment07>.

# Appendix

1. #------------------------------------------#
2. # Title: Assignment07.py
3. # Desc: Working with classes, functions, binary data and error handling.
4. # Change Log: (Who, When, What)
5. # CScott, 2021-Feb-23, Created File
6. # CScott, 2021-Feb-24, Revised DataProcessor functions
7. # CScott, 2021-Feb-28, Incorporated feedback and edits from Assignment06 comments
8. # CScott, 2021-Mar-01, Revised for Assignment07
9. #------------------------------------------#
10. **import** pickle
11. # -- DATA -- #
12. strChoice = '' # User input
13. lstTbl = []  # list of dictionaries to hold data
14. dicRow = {}  # dictionary of data row
15. strFileName = 'CDInventory.dat'  # data storage file
16. objFile = None  # file object
17. CD\_info = [None, None, None] # CD ID, title and artist
19. # -- PROCESSING -- #
20. **class** DataProcessor:
21. """Processing the data within the current inventory"""
23. @staticmethod
24. **def** add\_CD(CD\_info, table):
25. """Adds CD to current inventory
27. Args:
28. CD\_info (list): list of strgings containing CD ID, title and artist
29. table (list): list of dictionaries
30. Returns:
31. None
32. """
33. intID = int(CD\_info[0]) # Extracts intID from list CD\_info
34. strTitle = str(CD\_info[1]) # Extracts strTitle from list CD\_info
35. strArtist = str(CD\_info[2]) # Extracts strArtist from list CD\_info
36. dicRow = {'ID': intID, 'Title': strTitle, 'Artist': strArtist} # Creates dictionary containing all CD info
37. table.append(dicRow)
39. @staticmethod
40. **def** delete\_CD(intIDDel, table):
41. """Deletes a specific CD from current inventory based on ID
43. Args:
44. intIDDel (integer): ID of the CD to be deleted from current inventory
45. table (list): list of dictionaries
46. Returns:
47. blnCDRemoved (boolean): Boolean to track whether the specific CD ID was deleted
48. """
49. intRowNr = -1
50. blnCDRemoved = False
51. **for** row **in** table:
52. intRowNr += 1
53. **print** (intRowNr)
54. **if** row['ID'] == intIDDel:
55. **del** table[intRowNr]
56. blnCDRemoved = True
57. **break**
58. **return** blnCDRemoved

61. **class** FileProcessor:
62. """Processing the data to and from text file"""
64. @staticmethod
65. **def** read\_file(file\_name, table):
66. """Function to manage data ingestion from serialized file to a list of dictionaries
68. Reads the data from file identified by file\_name into a 2D table
69. (list of dicts) table one line in the file represents one dictionary row in table.
71. Args:
72. file\_name (string): name of file used to read the data from
73. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
75. Returns:
76. None.
77. """
78. **try**:
79. table.clear()  # this clears existing data and allows to load data from file
80. with open(file\_name, 'rb') as objFile:
81. pickled\_data = pickle.load(objFile)
82. **for** line **in** pickled\_data:
83. table.append(line)
84. **except** FileNotFoundError as e:
85. with open(file\_name, 'wb') as objFile:
86. **print**('ERROR:'+ e.\_\_doc\_\_ +' An empty file called ' + file\_name + ' was created within the working directory')
87. **except** EOFError as e:
88. **print**('The data file is empty! Current inventory is blank')

91. @staticmethod
92. **def** write\_file(file\_name, table):
93. """Function to manage exporting data from list of dictionaries to a serialized file
95. Writes the data to file identified by file\_name from a 2D table
96. (list of dicts) table one line in the file represents one dictionary row in table.
98. Args:
99. file\_name (string): name of file that the data is written to
100. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
102. Returns:
103. None.
104. """
105. with open(file\_name, 'wb') as file:
106. pickle.dump(table, file)

109. # -- PRESENTATION (Input/Output) -- #
111. **class** IO:
112. """Handling Input / Output"""
114. @staticmethod
115. **def** print\_menu():
116. """Displays a menu of choices to the user
118. Args:
119. None.
121. Returns:
122. None.
123. """
125. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
126. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
128. @staticmethod
129. **def** menu\_choice():
130. """Gets user input for menu selection
132. Args:
133. None.
135. Returns:
136. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
138. """
139. choice = ' '
140. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
141. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
142. **print**()  # Add extra space for layout
143. **return** choice
145. @staticmethod
146. **def** show\_inventory(table):
147. """
148. Displays current inventory table
150. Args:
151. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
153. Returns:
154. None.
156. """
157. **print**('======= The Current Inventory: =======')
158. **print**('ID\tCD Title (by: Artist)\n')
159. **for** row **in** table:
160. **print**('{}\t{} (by:{})'.format(\*row.values()))
161. **print**('======================================')
162. **print**() # Add extra space for layout
164. @staticmethod
165. **def** ID\_title\_artist\_add():
166. """Gets user input for specific CD to add
168. Args:
169. None.
171. Returns:
172. CD\_info (list): list containing CD ID, title and artist
174. """
175. **while** True:
176. **try**:
177. intID = int(input('Enter ID: ').strip())
178. **break**
179. **except** ValueError as e:
180. **print**('ERROR: ID that was entered is not of type: integer')
181. **print**(e.\_\_doc\_\_)
183. strTitle = input('What is the CD\'s title? ').strip()
184. strArtist = input('What is the Artist\'s name? ').strip()
185. **print**() # Add extra space for layout
186. CD\_info = [intID, strTitle, strArtist]
187. **return** CD\_info
189. # 1. When program starts, read in the currently saved Inventory
190. FileProcessor.read\_file(strFileName, lstTbl)
192. # 2. Start main loop
193. **while** True:
194. # 2.1 Display Menu to user and get choice
195. IO.print\_menu()
196. strChoice = IO.menu\_choice()
197. # 3. Process menu selection
198. # 3.1 process exit first
199. **if** strChoice == 'x':
200. **break**
201. # 3.2 process load inventory
202. **if** strChoice == 'l':
203. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
204. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
205. **if** strYesNo.lower() == 'yes':
206. **print**('reloading...')
207. FileProcessor.read\_file(strFileName, lstTbl)
208. IO.show\_inventory(lstTbl)
209. **else**:
210. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
211. IO.show\_inventory(lstTbl)
212. **continue**  # start loop back at top.
213. # 3.3 process add a CD
214. **elif** strChoice == 'a':
215. # 3.3.1 Ask user for new ID, CD Title and Artist
216. CD\_info=IO.ID\_title\_artist\_add()
217. # 3.3.2 Add item to the table
218. DataProcessor.add\_CD(CD\_info, lstTbl)
219. # 3.3.3 Display current inventory after adding CD
220. IO.show\_inventory(lstTbl)
221. **continue**  # start loop back at top.
222. # 3.4 process display current inventory
223. **elif** strChoice == 'i':
224. IO.show\_inventory(lstTbl)
225. **continue**  # start loop back at top.
226. # 3.5 process delete a CD
227. **elif** strChoice == 'd':
228. # 3.5.1 get Userinput for which CD to delete
229. # 3.5.1.1 display Inventory to user
230. IO.show\_inventory(lstTbl)
231. # 3.5.1.2 ask user which ID to remove
232. **try**:
233. intIDDel = int(input('Which ID would you like to delete? ').strip())
234. # 3.5.2 search thru table and delete CD
235. removed = DataProcessor.delete\_CD(intIDDel,lstTbl)
236. **if** removed:
237. **print**('The CD was removed')
238. **else**:
239. **print**('Could not find this CD!')
240. **except** ValueError:
241. **print** ('Please enter an integer if you would like to delete an entry \n')
242. # 3.5.3 display current inventory after deleting CD
243. IO.show\_inventory(lstTbl)
244. **continue**  # start loop back at top.
245. # 3.6 process save inventory to file
246. **elif** strChoice == 's':
247. # 3.6.1 Display current inventory and ask user for confirmation to save
248. IO.show\_inventory(lstTbl)
249. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
250. # 3.6.2 Process choice
251. **if** strYesNo == 'y':
252. # 3.6.2.1 save data
253. FileProcessor.write\_file(strFileName, lstTbl)
254. **else**:
255. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
256. **continue**  # start loop back at top.
257. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be safe:
258. **else**:
259. **print**('General Error')

Listing 1. CDInventory.py code as formatted from <http://www.planetb.ca/syntax-highlight-word>