Brandon Wayward

2020-Aug-23

IT FDN 110 B SU 20

Assignment 07

Adventures in Error Handling, Pickling, and other dark arts.

Introduction

In module 7 we’ve learned about working with text files and binary files in addition to learning how to use structured error handling

# Working with Text files

We’ve already been familiarized with text files in the past few lessons with regards on how to write to a file, read a file, and append data to that file, but when using those methodologies within a function it allows all of the file operations (open, read/write/append, close) to take place within the function instead of being repeated multiple times throughout your codebase.

When reading files, we formally only knew about reading the entire file as a whole, but this lesson introduced us to two additional methodologies. The readline() function and the readlines() function

The readline() function, one each call, returned the next singe line of data in a file. While the access to the file stays open, the location of which line to read next is memorized. This caught me off-guard once as I didn’t iterate through the lines and closed the file. Leave the file open, and python will remember where it is so it can call the next line. This function shows primary use cases in for and while loops so that it can iterate through a file.

The readlines() function reads all the lines in a file and returns a list (Whereas the read() function returns a string). Readlines() includes everything in the line, new line chars, escape char, overloads, etc.

# Working with Binary Files

Up until this point we’ve primarily saved data in text files that are human readable. This isn’t the same way the information is stored in memory. Additionally, with text files we’ve had to perform formatting steps to allow the data to be stored and retrieved a certain way. These additional steps require more processing time and memory. If these additional steps aren’t truly necessary, the easiest solution is to just save the information as it is in memory. This is the binary code and it’s an incredible practical way to save states of your application or setting the user made.

# Pickling

Saving binary information in Python is called pickling. The **pickle** module takes the information associated with an object and serializes / de-serializes it in a way that it can be easily stored / loaded as binary information

Pickling also has a wide variety of use cases [[1]](#footnote-1) You can utilize it with machine learning to store models so you don’t have to retrain them later, send data over tcp or socket (really?!) or what we’ve learned here, save your programs state data to disk. Pickling should not be used when saving data across multiple languages as it’s a Python specific protocol. This also holds true for different versions of Python, as the standard may change from version to version!

A Very clear and concise example and tutorial for utilizing pickling is available at:

[Na-Rae Han's Python 3 Notes](https://www.pitt.edu/~naraehan/python3/pickling.html)

# Exception Handling

Up until now, when we coded something Python didn’t like or made small mistakes in our code, the script would end and crash with an exception. With that in mind, it would make sense to protect parts of the program with structured error handling whenever there is an understanding that something may go wrong. We can solve for this by setting up try, else statement traps to catch the errors, handle the exceptions, and customize how the program handles those errors.

While we only directly discuss “try / except” error handling, over at RealPython[[2]](#footnote-2) their very well crafted tutorial mentions other functions such as raise (to force an exception of a specific kind) and assert (to test if a condition is true or not). Building on try/ except, you can add an else statement after the except. The added Else basically says “If there is no exception, run this code!” I was kind of amazed that you could add up to three levels in that statement and then they have to go ahead and drop finally on us. In addition to a codeblock with try, except, and else, you can then close it up with finally. Finally code should be blocks of code that you always want ran with/without exceptions ever taking place.

Breaking it down as follows

Try -> run this code

Except -> Execute on exception

Else -> No exceptions? Run this

Finally -> Always run this code

# Summary

In summary, we discovered how to increase the armor class on our scripts with exception handling so that if we or the user do something unexpected the script will alert us and either keep moving or gracefully exit. We also learned how to write to and read from binary files, I still don’t understand why it’s called pickling though!

# Appendix

Github: <https://github.com/itsallparticles/Assignment_07>

Code:

1. #------------------------------------------#
2. # Title: CDInventory.py
3. # Desc: Working with classes and functions.
4. # Change Log: (Who, When, What)
5. # DBiesinger, 2030-Jan-01, Created File
6. # BWayward, 2020-Aug-15 Created write\_file and remove\_inventory function, added DocStrings to both
7. # BWayward, 2020-Aug-16 Cleaned up write to file and Delete codebase, attempted to create add\_inventory function(Failed)
8. # BWayward, 2020-Aug-17 Revisited add inventory function. added IO.addCD and DataProcessor.add\_inventory
9. # BWayward, 2020-Aug-18 After class, added file check to ensure text file exists.
10. # DKlos, 2020-Aug-20, Corrections
11. # BWayward, 2020-Aug-21, modified txt to .dat, import pickle, Added information to read and write functions re: Binary file handling
12. # BWayward, 2020-Aug-22, added error handling for file missing and int check for deletion request
13. #------------------------------------------#
14. **import** pickle

17. # -- DATA -- #
19. strChoice = '' # User input
20. lstTbl = []  # list of lists to hold data
21. dicRow = {}  # list of data row
22. strFileName = 'CDInventory.dat'  # data storage file
23. objFile = None  # file object
25. # -- PROCESSING -- #
26. **class** DataProcessor:
28. @staticmethod
29. **def** remove\_inventory(selectID, table):
30. """ Function to manage removal of items from table in memory
32. Args:
33. None
35. Returns:
36. prints 'The CD was removed', and displays current inventory with item deleted
37. """
38. intRowNr = -1
39. blnCDRemoved = False
40. **for** row **in** table:
41. intRowNr += 1
42. **if** row['ID'] == selectID:
43. **del** table[intRowNr]
44. blnCDRemoved = True
45. **break**
46. **if** blnCDRemoved:
47. **print**('The Compact Disc was removed')
48. **else**:
49. **print**('Could not find this Compact Disc!')
51. @staticmethod
52. **def** add\_inventory(strID, strTitle, strArtist, table):
53. """ Processing user input into list of dicts
55. Args:
56. userInput: Conversion of strID from string to intenger.
57. dictionary: dictionary where the user input data is stored
58. table: Table where dictionary is written
59. Returns:
60. None """
61. intID = int(strID)
62. dicRow = {'ID': intID, 'Title': strTitle, 'Artist': strArtist}
63. table.append(dicRow)

66. **class** FileProcessor:
67. """Processing the data to and from text file"""
69. @staticmethod
70. **def** read\_file(file\_name, table):
71. """Function to manage data ingestion from file to a list of dictionaries
73. Reads the data from file identified by file\_name into a 2D table
74. (list of dicts) table one line in the file represents one dictionary row in table.
76. Args:
77. file\_name (string): name of file used to read the data from
78. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
80. Returns:
81. None.
82. """
83. table.clear()  # this clears existing data and allows to load data from file
85. **try**:
86. objFile = open(file\_name, 'rb') #Reads from binary file in.
87. # Retrieve the preserved data (list of dictionaries) from previous write session
88. data = pickle.load(objFile)
89. table.extend(data)
90. objFile.close()
91. **except** FileNotFoundError as e:
92. **print**("\nNo Database file currently exists.  Please add and save compact disc information to create a file.\n")
93. **print**(type(e), e, e.\_\_doc\_\_, sep='\n')
95. @staticmethod
96. **def** write\_file(file\_name, table):
97. """ Function to write data to pickled file.
99. Writes the data to a specified file identified by file\_name and 2d table.
101. Args:
102. file\_name (string): name of file used to read the data from
103. table (list of dict): 2D data structure tha holds data during runtime
105. Returns:
106. None.
107. """
109. objFile = open(file\_name, 'wb')
110. pickle.dump(table, objFile)
111. objFile.close()


115. # -- PRESENTATION (Input/Output) -- #
117. **class** IO:
118. """Handling Input / Output"""
120. @staticmethod
121. **def** print\_menu():
122. """Displays a menu of choices to the user
124. Args:
125. None.
127. Returns:
128. None.
129. """
131. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
132. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
134. @staticmethod
135. **def** menu\_choice():
136. """Gets user input for menu selection
138. Args:
139. None.
141. Returns:
142. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
144. """
145. choice = ' '
146. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
147. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
148. **print**()  # Add extra space for layout
149. **return** choice
151. @staticmethod
152. **def** show\_inventory(table):
153. """Displays current inventory table

156. Args:
157. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
159. Returns:
160. None.
162. """
163. **print**('======= The Current Inventory: =======')
164. **print**('ID\tCD Title (by: Artist)\n')
165. **for** row **in** table:
166. **print**('{}\t{} (by:{})'.format(\*row.values()))
167. **print**('======================================')
169. # Missing staticmethod
170. @staticmethod
171. **def** addCD():
172. """ Process user input to add data
174. Args:
175. none
176. Returns:
177. none
178. """
180. # This is all good.
181. strID = input('Enter ID: ').strip()
182. strTitle = input('What is the CD\'s title? ').strip()
183. stArtist = input('What is the Artist\'s name? ').strip()
184. **return** strID, strTitle, stArtist
186. # 1. When program starts, read in the currently saved Inventory
187. FileProcessor.read\_file(strFileName, lstTbl)

190. # 2. start main loop
191. **while** True:
192. # 2.1 Display Menu to user and get choice
193. IO.print\_menu()
194. strChoice = IO.menu\_choice()
196. # 3. Process menu selection
197. # 3.1 process exit first
198. **if** strChoice == 'x':
199. **break**
200. # 3.2 process load inventory
201. **if** strChoice == 'l':
202. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
203. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
204. **if** strYesNo.lower() == 'yes':
205. **print**('reloading...')
206. FileProcessor.read\_file(strFileName, lstTbl) #Calls function that loads text file containing CD inventory into runtime.
207. IO.show\_inventory(lstTbl) #Calls function that displays inventory to user
208. **else**:
209. input('cancelling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
210. IO.show\_inventory(lstTbl) #Calls function that displays inventory to user
211. **continue**  # start loop back at top.
212. # 3.3 process add a CD
213. **elif** strChoice == 'a':
214. # 3.3.1 Calls function that asks user for new ID, CD Title and Artist
215. strID, strTitle, strArtist = IO.addCD()
216. # 3.3.2 Calls the function that adds item to the table
217. DataProcessor.add\_inventory(strID, strTitle, strArtist, lstTbl)
218. #Calls function that displays inventory with added CD
219. IO.show\_inventory(lstTbl)
220. **continue**  # start loop back at top.
221. # 3.4 process display current inventory
222. **elif** strChoice == 'i':
223. IO.show\_inventory(lstTbl) # Calls function that displays current inventory
224. **continue**  # start loop back at top.
225. # 3.5 process delete a CD
226. **elif** strChoice == 'd':
227. # 3.5.1 get Userinput for which CD to delete
228. # 3.5.1.1 Calls function that displays inventory to user
229. IO.show\_inventory(lstTbl)
230. # 3.5.1.2 ask user which ID to remove
231. **try**: intIDDel = int(input('Which ID would you like to delete? ').strip())
232. **except** ValueError:
233. **print**("\nYou must enter ID as a integer. Deleting CD failed.\n")
234. # 3.5.2 Calls function that searches thru table and deletes CD
235. **else**:
236. DataProcessor.remove\_inventory(intIDDel, lstTbl)
237. IO.show\_inventory(lstTbl) #Calls function that displays inventory to user
238. **continue**  # start loop back at top.
239. # 3.6 process save inventory to file
240. **elif** strChoice == 's':
241. # 3.6.1 Calls function that displays current inventory.
242. IO.show\_inventory(lstTbl)
243. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower() #asks user for confirmation to save
244. # 3.6.2 Process choice
245. **if** strYesNo == 'y':
246. # 3.6.2.1 Calls function that saves data
247. FileProcessor.write\_file(strFileName, lstTbl)
248. **else**:
249. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
250. **continue**  # start loop back at top.
251. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be save:
252. **else**:
253. **print**('General Error')

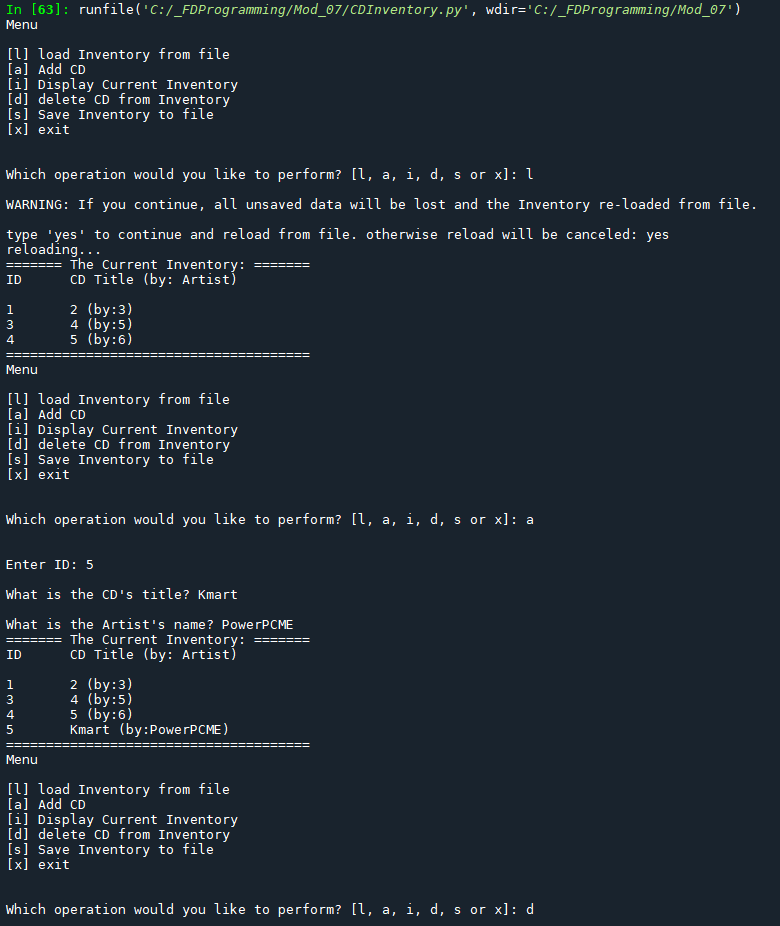


Figure - Script running in Spider, 1

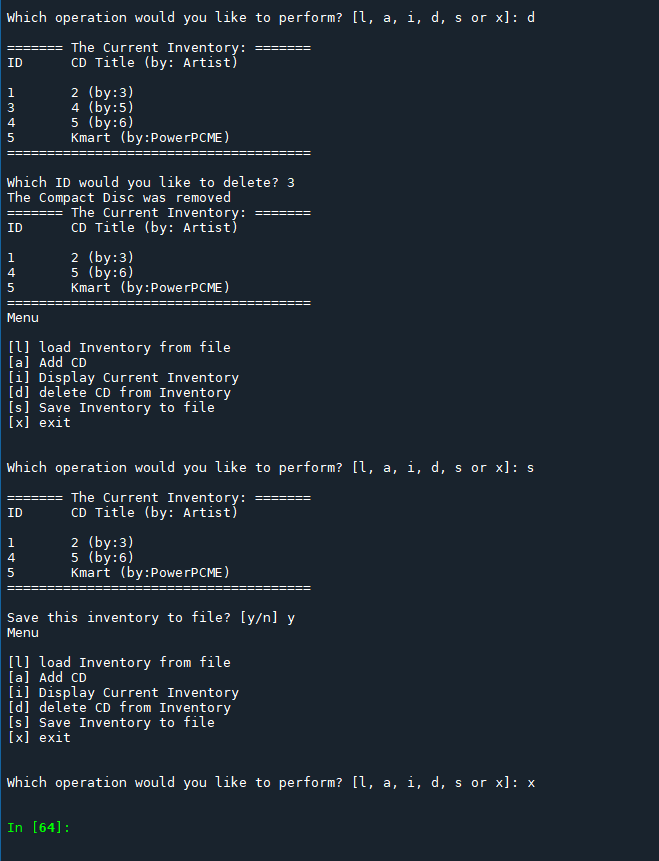


Figure - Script running in spider 2

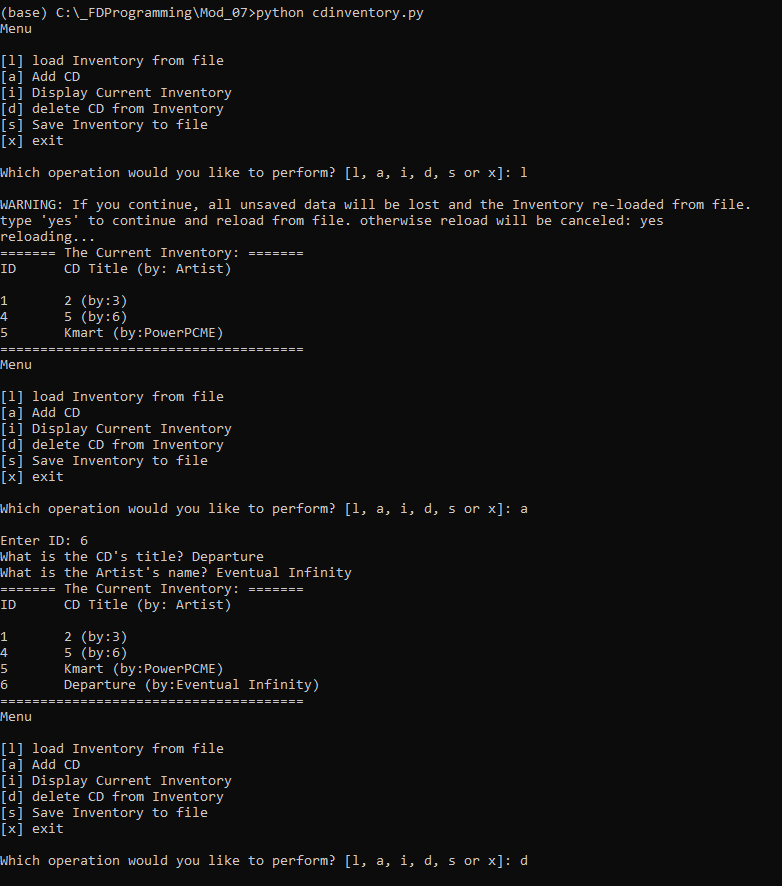


Figure - Script in Terminal 1

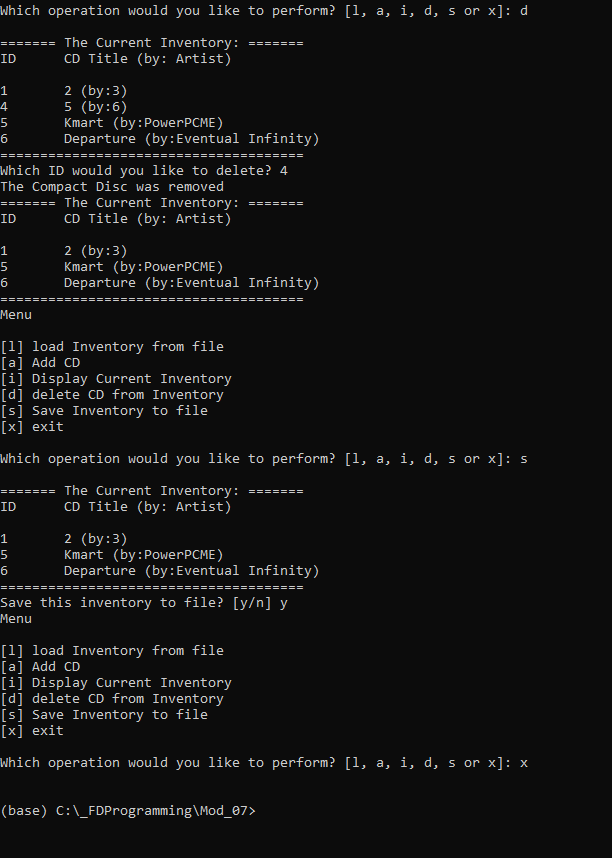


Figure - Script running as terminal 2

1. <https://www.datacamp.com/community/tutorials/pickle-python-tutorial> [↑](#footnote-ref-1)
2. <https://realpython.com/python-exceptions/> [↑](#footnote-ref-2)