Britta Anson

August 26, 2020

Foundations of Programming: Python

Assignment 07

Modified CD Inventory Script:

Structured Error Handling

# Introduction

In module 7 we learned about structured error handling, different ways to work with text files and binary files (including “pickling”) and were introduced to markdown language.

# General background

## What are the benefits of using structured error handling?

Structured error handling allows us to remain in control of a program regardless of user input. It is a way to catch and provide feedback on anticipated user errors and to handle those errors in a way that doesn’t crash the program. Handling errors in a controlled manner saves the user from unexpectedly losing their data by crashing the program and creates a more user-friendly experience.[[1]](#footnote-1)

## What are the differences between a text file and a binary file?

A text file is easily readable by a human whereas a binary file is not. A binary file stores data in the same format in which it exists in memory.[[2]](#footnote-2)

## Pickling

“Pickling” is the term for saving binary information in Python. This can be done using the **pickle module**.[[3]](#footnote-3)

## How is the Exception class used?

The Exception class is a class built into Python that holds information about an error. It is used by creating exception objects each time an error occurs, which can be used to provide information about that error to the user. The exception object is often referred to as ‘e’ within a script.[[4]](#footnote-4)

## How do you "derive" a new class from the Exception class?

Besides built-in child classes of the Except class (such as “ValueError” or “ZeroDivisionError” etc.) it is possible to create custom child classes of the Exception class. These classes are derived from the base class and can build upon data and functions inherited from the base class.[[5]](#footnote-5)

These custom Exception classes are defined by naming them with “Exception” in parentheses after the custom name. This signals that the class is building upon the “Exception” base class.[[6]](#footnote-6)

## When might you create a class derived from the Exception class?

You might create a custom class derived from the Exception class if you want to define custom errors that would not automatically be viewed as an error, such as setting parameters that an input must meet certain specifications (like within a specific value range, etc.).

## What is the Markdown language?

The Markdown language is “a plain text format for writing structured documents, based on conventions for indicating formatting in email and usenet posts.”[[7]](#footnote-7) This is what we would use for creating readme files in GitHub.

# Modifying the CD Inventory script

This week we were not given fresh starter code, but rather were asked to build on our Assignment 06 version of the CDInventory. Our task this week was to:

1. Add structured error handling (using built-in Python error classes) around areas where there is:
   1. user interaction
   2. type casting (string to int); or
   3. file access operations
2. Modify the permanent data store to use binary data

I started with the edited version received back from Douglas Klos in which he had removed dependence on global variables in some of my functions and added the table to the function arguments, which I had left out. I updated the docstrings to include the table as an argument. After he did that and I better understood the process of abstraction, I realized that I was reusing the same variable names (strID, strTitle, strArtist) in my add\_data() function as in my program and that they were actually not referencing the same things so it would be clearer to differentiate those names. I started by renaming the add\_data() arguments to the abstract “ID”, “title” and “artist” to differentiate them from those that are ultimately passed into the function in the program.

Screen of a cell phone

Description automatically generated

Listing 1 - add\_data function from Doug's edited version of Assignment 06 version of CDInventory.py

A screen shot of a smart phone

Description automatically generated

Listing 2 - Updated add\_data function from Assignment 07 version of CDInventory.py (now with abstract argument names)

Next I went through the script and added TODO items based on the assignment instructions.

## Adding Error Handling

I found three places to add error handling:

1. In the type casting of string to integer in the add\_data function
2. In case of entry that can’t be converted to integer in get\_data function
3. In case file doesn’t exist in attempting to access file in read\_file function

Several of these cases are redundant in the context of this program. For example, this program starts by calling the read\_data() function and, if the specified file does not exist, creates that file. Therefore, it is really unnecessary to check for errors in the read\_data functions. However, I’m assuming that since the functions are meant to be self-contained and should be able to be copied into another script, we cannot assume that the errors would be irrelevant in another environment.

Similarly, there is redundancy in checking that the ID value can be converted to integer at both the point it is entered by the user, and the point it is passed into the add\_data function.

### Type-casting error

In the add\_data function, I added a try-except block with a ValueError exception in case the conversion of the ID value to integer throws an error:

A screenshot of a cell phone

Description automatically generated

Listing 3 - Error handling added to add\_data function

A screenshot of a cell phone

Description automatically generated

Figure 1 - Program running in Spyder, showing correct implementation of the error for a non-integer ID

This iteration is awkward because it allows the data entry to continue and only throws the type error after the user confirms the data and decides to add it to the table. That’s because this menu selection ‘a’ calls two separate functions back-to-back: the ﻿IO.get\_data() function, and the DataProcessor.add\_data() function. Once the error handling is added to the first function, it will be irrelevant in the second (and impossible to test), which is why I added error handling to the second-called function first (to ensure that the function is self-contained and would include the correct error handling if moved to another program).

### User interaction error

Next I updated the IO.get\_data() function to check that the user entered ID is a number that can be converted to an integer. I used a while loop so that if a non-integer is entered by the user, it will repeat the prompt to enter an ID after throwing the error.

A screenshot of a cell phone

Description automatically generated

Listing 4 - get\_data() function with error handling of the user input "ID" to make sure it is a number

Spyder throws up an alert indicating that the variable “intID” is defined but never used, but it does not prevent the program from running and I never intended to use it other than to check the input was a number. I suspect this may not be best practice (?) but I’m not sure.

A screenshot of a cell phone

Description automatically generated

Figure 2 - The error handling working correctly in Spyder console.

### handling data accessing error

Next I added a FileNotFoundError exception to the read\_file() function. The write\_file() function doesn’t need one because if the file called in the write\_file() function doesn’t exist, it will create one.

First, in order to be able to test this addition as I go, I commented out the code at the beginning of the program where it already accounts for a missing file name. This way I test the error handling in the function.

A screenshot of a cell phone

Description automatically generated

Listing 5 - Commented out handling of missing file built into program, in order to test error handling in read\_file() function

A screenshot of a cell phone

Description automatically generated

Listing 6 - Updated read\_file() function including error handling of missing file

Running the program in Spyder console:

A screenshot of a cell phone

Description automatically generated

Figure 3 - This shows the error message at the start of the program when it fails to find the designated file to read. It allows the program to continue without the file info (a user could just create new data and write a new file if they choose)

## Modifying data store to use binary data

Next, to change the permanent storage from text to binary storage, I first made a pass through the script and added TODO items in all places that referenced the text file, with a note to change to binary storage. This included:

1. Adding “import pickle” to start of script
2. Create new variable "binFileName" for binary data file
3. Rewrite read\_file() and write\_file() functions to pickle data and work with binary file “CDInventory.dat”
4. Change all instances of strFileName to binFileName when used as argument passed into a function call (in the ‘l’ (load) and ‘s’ (save) menu choices)

I rewrote the read\_file() and write\_file() functions based on the examples “read\_data()” and “save\_data()” in Listing 9 of the Module 07 document.[[8]](#footnote-8)

# RUNNING the FINAL code

## Running in Spyder

A screenshot of a cell phone

Description automatically generated

Figure 4 - CDInventory.py running in Spyder console (part 1)

The program prompts the user to first load the data into the table after reading the file at the start.

A close up of text on a black background

Description automatically generated

Figure 5 - CDInentory.py running in Spyder console (part 2)

This example shows the error handling in the event that a user enters non-integer text into the ID field. It also shows successful add of new CD and save.

## Running in the Terminal

A screenshot of a social media post

Description automatically generated

Figure 6 - CDInventory.py running in Terminal (part 1)

This instance shows the saved data from the last run (in Spyder) successfully loaded by the program in the Terminal. It also shows successful delete feature.

A screenshot of a cell phone

Description automatically generated

Figure 7 - CDInventory.py running in Terminal (part 2)

This continuation shows adding a new CD to replace the one that was deleted, and exiting the program.

# Posted to github

The assignment documents are posted to my GitHub repository “Assignment\_07” at: <https://github.com/bricolagerie/Assignment_07>.

# Summary

Changing the data storage from a text file to a binary file required tweaking some functions throughout the script, but ultimately it was a lot simpler to handle because reading and writing didn’t require any special processing. Error handling was simpler in some situations than others, and I will need more practice to become more adept at how to implement this technique.

# Appendix

## Listing CDInventory.py

1. #------------------------------------------#
2. # Title: CDInventory.py
3. # Desc: Assignment 07 - Working with classes and functions.
4. # Change Log: (Who, When, What)
5. # BAnson, 2020-Aug-16, Created File for Assignment 06
6. # BAnson, 2020-Aug-16, moved write to file code to write\_file function
7. #   Added try-except block to create new file if none exists
8. #   Cleaned up formatting in I/O presentation
9. #   Added data processing functions add\_data and del\_id
10. #   Added IO functions to collect user inputs of new\_id, new\_title, new\_artist
11. # BAnson, 2020-Aug-18, removed calls to IO class from DataProcessor class in add\_data()
12. #   Consolidated 3 user input IO functions into 1 get\_data()
13. #   Added:
14. #       Review of entered data and y/n choice to add to table
15. #       "are you sure?" to exit sequence
16. # DKlos, 2020-Aug-21, Removed dependence on global variables.
17. # BAnson, 2020-Aug-25, resaved for Assignment 07
18. # BAnson, 2020-Aug-26, renamed add\_data args to be distinct from program variables;
19. #       Updated docstrings.
20. #       Added error handling to add\_data(), read\_file(), and get\_data()
21. #       Recreated read\_file() and write\_file() using pickling
22. #------------------------------------------#
24. # TODid: import pickle
25. **import** pickle
27. # -- DATA -------------------------------------------------------------------- #
29. strChoice = '' # User input
30. lstTbl = []  # list of lists to hold data
31. dicRow = {}  # list of data row
32. # TODid: Create new binary file variable for data storage
33. binFileName = 'CDInventory.dat'  # binary data storage file
34. objFile = None  # file object


38. # -- PROCESSING FUNCTIONS ---------------------------------------------------- #
40. **class** DataProcessor:
42. @staticmethod
43. **def** add\_data(ID, title, artist, table):
44. """Appends Inventory with values assigned to ID, title, and artist
46. Args:
47. ID (string) = user input ID
48. title (string) = user input Title
49. artist (string) = user input Artist
50. table of dicts = user designated table to store inventory
52. Returns:
53. Integer version of ID (intID)
54. A dictionary row (dictRow) containing intID, title, artist
56. """
57. # TODid: added error handling for type casting
58. # In the context of this script this is redundant because this is already
59. #   checked in the get\_data() function, but the intention is to make
60. #   this function self-contained
61. **try**:
62. intID = int(ID)
63. dicRow = {'ID': intID, 'Title': title, 'Artist': artist}
64. table.append(dicRow)
65. **except** ValueError as e:
66. **print**('That ID is not an integer!')
67. **print**('Build in error info:')
68. **print**(type(e), e, e.\_\_doc\_\_, sep= '\n')

71. @staticmethod
72. **def** del\_id(id\_to\_delete, table):
73. """Deletes CD inventory item based on ID
75. Args:
76. id\_to\_delete = user input ID
77. table of dicts = user designated table from which to delete row
79. Returns:
80. None
82. """
83. intRowNr = -1
84. blnCDRemoved = False
85. **for** row **in** table:
86. intRowNr += 1
87. **if** row['ID'] == id\_to\_delete:
88. **del** table[intRowNr]
89. blnCDRemoved = True
90. **break**
91. **if** blnCDRemoved:
92. **print**('The CD was removed')
93. **else**:
94. **print**('Could not find this CD!')

97. **class** FileProcessor:
98. """Processing the data to and from text file"""
100. # TODid: Create read\_file to read from binary file
101. @staticmethod
102. **def** read\_file(file\_name):
103. """Function to manage data ingestion from file to a list of dictionaries
105. Reads the data from file identified by file\_name into a 2D table
106. (list of dicts) table one line in the file represents one dictionary row in table.
108. Args:
109. file\_name (string): name of file used to read the data from
111. Returns:
112. table (list of dicts): 2D data structure (list of dicts) that holds the data during runtime
113. """
115. # TODid: Add error handling in case file doesn't exist
116. **try**:
117. with open(file\_name, 'rb') as fileObj:
118. table = pickle.load(fileObj)
119. **print**('File read complete. Load data to populate table.')
120. **return** table
121. **except** FileNotFoundError as e:
122. **print**('This file does not exist!')
123. **print**('Build in error info:')
124. **print**(type(e), e, e.\_\_doc\_\_, sep= '\n')
125. **print**('\nAdd data and save to create file.')

128. # TODid: Create write\_file function to write to binary file
129. @staticmethod
130. **def** write\_file(table, file\_name):
131. """Writes data in memory table to a binary file (or creates binary text file if none exists
133. Args:
134. file\_name = name of the file to open and write data to (or create)
135. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
137. Returns:
138. Binary file containing data of current table in memory
139. """
140. with open(file\_name, 'wb') as fileObj:
141. pickle.dump(table, fileObj)

144. # -- PRESENTATION (Input/Output) FUNCTIONS ----------------------------------- #
146. **class** IO:
147. """Handling Input / Output"""
149. @staticmethod
150. **def** print\_menu():
151. """Displays a menu of choices to the user
153. Args:
154. None.
156. Returns:
157. None.
158. """
160. **print**('\nMenu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
161. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
163. @staticmethod
164. **def** menu\_choice():
165. """Gets user input for menu selectioni
167. Args:
168. None.
170. Returns:
171. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
173. """
174. choice = ' '
175. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
176. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
177. **print**()  # Add extra space for layout
178. **return** choice
180. @staticmethod
181. **def** show\_inventory(table):
182. """Displays current inventory table
184. Args:
185. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
187. Returns:
188. None.
190. """
191. **print**('\n======= The Current Inventory: =======')
192. **print**('ID\tCD Title (by: Artist)\n')
193. **for** row **in** table:
194. **print**('{}\t{} (by: {})'.format(\*row.values()))
195. **print**('======================================')
197. # added I/O functions:
199. @staticmethod
200. **def** get\_data():
201. """Collects three pieces of data from user: ID, title, and artist
203. Args:
204. None
206. Returns:
207. Tuple containing values corresponding to ID (strID), title (strTitle), and artist (strArtist)
209. """
210. # TODid: Add error handling to make sure entry can be converted to integer
211. **while** True:
212. strID = input('Enter ID: ').strip()
213. **try**:
214. intID = int(strID)
215. **break**
216. **except** ValueError as e:
217. **print**('That ID is not an integer!')
218. **print**('Build in error info:')
219. **print**(type(e), e, e.\_\_doc\_\_, sep= '\n')
220. strTitle = input('What is the CD\'s title? ').strip()
221. strArtist = input('What is the Artist\'s name? ').strip()
222. tplUserData = (strID, strTitle, strArtist)
223. **return** tplUserData

226. # -- MAIN PROGRAM ------------------------------------------------------------ #
228. # 1. When program starts, read in the currently saved Inventory, or create empty file
229. # TODid: change file name variable to binFileName
230. **try**:
231. FileProcessor.read\_file(binFileName)
233. **except**:
234. **print**('\nUse \'a\' to add new data to the table and \'s\' to save and create a new file.')

237. # 2. start main loop
238. **while** True:
239. # 2.1 Display Menu to user and get choice
240. IO.print\_menu()
241. strChoice = IO.menu\_choice()
243. # 3. Process menu selection
244. # 3.1 process exit first
245. **if** strChoice == 'x':
246. choice = input('Are you sure you want to exit? [y/n]: ') # prevent exiting accidentally
247. **if** choice.lower() == 'y':
248. **break**
249. **else**:
250. **continue**
252. # 3.2 process load inventory
253. **if** strChoice == 'l':
254. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
255. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
256. **if** strYesNo.lower() == 'yes':
257. lstTbl.clear()
258. **print**('reloading...')
259. # TODid: change strFileName to binFileName
260. lstTbl = FileProcessor.read\_file(binFileName)
261. **if** lstTbl != None:
262. IO.show\_inventory(lstTbl)
263. **else**:
264. lstTbl = []
265. IO.show\_inventory(lstTbl)
266. **else**:
267. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
268. IO.show\_inventory(lstTbl)
269. **continue**  # start loop back at top.
271. # 3.3 process add a CD
272. **elif** strChoice == 'a':
273. # 3.3.1 Ask user for new ID, CD Title and Artist
274. # Moved IO code into function
275. tplUserData = IO.get\_data()
276. # Review entry
277. **print**('\nYou entered: ', tplUserData)
278. choice = input('Continue to add data to table? [y/n]: ')
279. **if** choice.lower() == 'y':
280. strID, strTitle, strArtist = tplUserData
281. # 3.3.2 Add item to the table
282. # Need to pass lstTbl to the function
283. # DataProcessor.add\_data(strID, strTitle, strArtist)
284. DataProcessor.add\_data(strID, strTitle, strArtist, lstTbl)
285. IO.show\_inventory(lstTbl)
286. **else**:
287. **print**('Data not added to table.')
288. **continue**  # start loop back at top.
290. # 3.4 process display current inventory
291. **elif** strChoice == 'i':
292. **if** lstTbl:
293. IO.show\_inventory(lstTbl)
294. **else**:
295. **print**('Inventory is empty. Use \'l\' to load existing inventory or \'a\' to add CD to new inventory.')
296. **continue**  # start loop back at top.
298. # 3.5 process delete a CD
299. **elif** strChoice == 'd':
300. # 3.5.1 get Userinput for which CD to delete
301. # 3.5.1.1 display Inventory to user
302. IO.show\_inventory(lstTbl)
303. # 3.5.1.2 ask user which ID to remove
304. intIDDel = int(input('Which ID would you like to delete? ').strip())
305. # 3.5.2 search thru table and delete CD
306. # Moved processing code into function
307. # Need to pass in id to delete and table to delete from.
308. # DataProcessor.del\_id()
309. DataProcessor.del\_id(intIDDel, lstTbl)
310. IO.show\_inventory(lstTbl)
311. **continue**  # start loop back at top.
313. # 3.6 process save inventory to file
314. **elif** strChoice == 's':
315. # 3.6.1 Display current inventory and ask user for confirmation to save
316. IO.show\_inventory(lstTbl)
317. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
318. # 3.6.2 Process choice
319. **if** strYesNo == 'y':
320. # 3.6.2.1 save data
321. # TODid: Change strFileName to binFileName
322. FileProcessor.write\_file(lstTbl, binFileName) # function call replaces previous code
323. **else**:
324. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
325. **continue**  # start loop back at top.
327. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be save:
328. **else**:
329. **print**('General Error')

1. Dirk Biesinger, Module 07 Document, pp. 16-18 [↑](#footnote-ref-1)
2. Module 07 Doc, p. 14 [↑](#footnote-ref-2)
3. Module 07 Doc, p. 14 [↑](#footnote-ref-3)
4. Module 07 Doc, p. 18-19 [↑](#footnote-ref-4)
5. Module 07 Doc, pp. 21-22 [↑](#footnote-ref-5)
6. Module 07 Doc, p. 23 [↑](#footnote-ref-6)
7. “GitHub Flavored Markdown Spec,” <https://github.github.com/gfm/> (Retrieved, 2020-Aug-26) [↑](#footnote-ref-7)
8. Module 07 Doc, p. 15 [↑](#footnote-ref-8)