Joseph Prado

23 March 2020

## IT FDN 100 B

Assignment 07

**Pickling and Structured Exception Handling**

# Introduction

In this assignment, I will do several things.

1. Identify websites that are good at explaining Python’s Exception Handling.
2. Identify websites that are good at explaining Python’s pickling feature.
3. Modify an existing script to utilize structured exception handling around area where there is user interaction, type casting, or file access operations.
4. Modify an existing script to utilize the pickling feature to read and write from binary files.

# Identifying websites that are good at explaining Python’s Exception handling

### **Website 1:** [Data Camp “Exception and Error Handling in Python” Tutorial](https://www.datacamp.com/community/tutorials/exception-handling-python)[[1]](#footnote-1)

**Why is the website good?:**

1. **Provides outstanding visual depiction at the beginning that describes how “try”, “except” “else” and “finally” statements work with each other to handle exceptions. See below.**

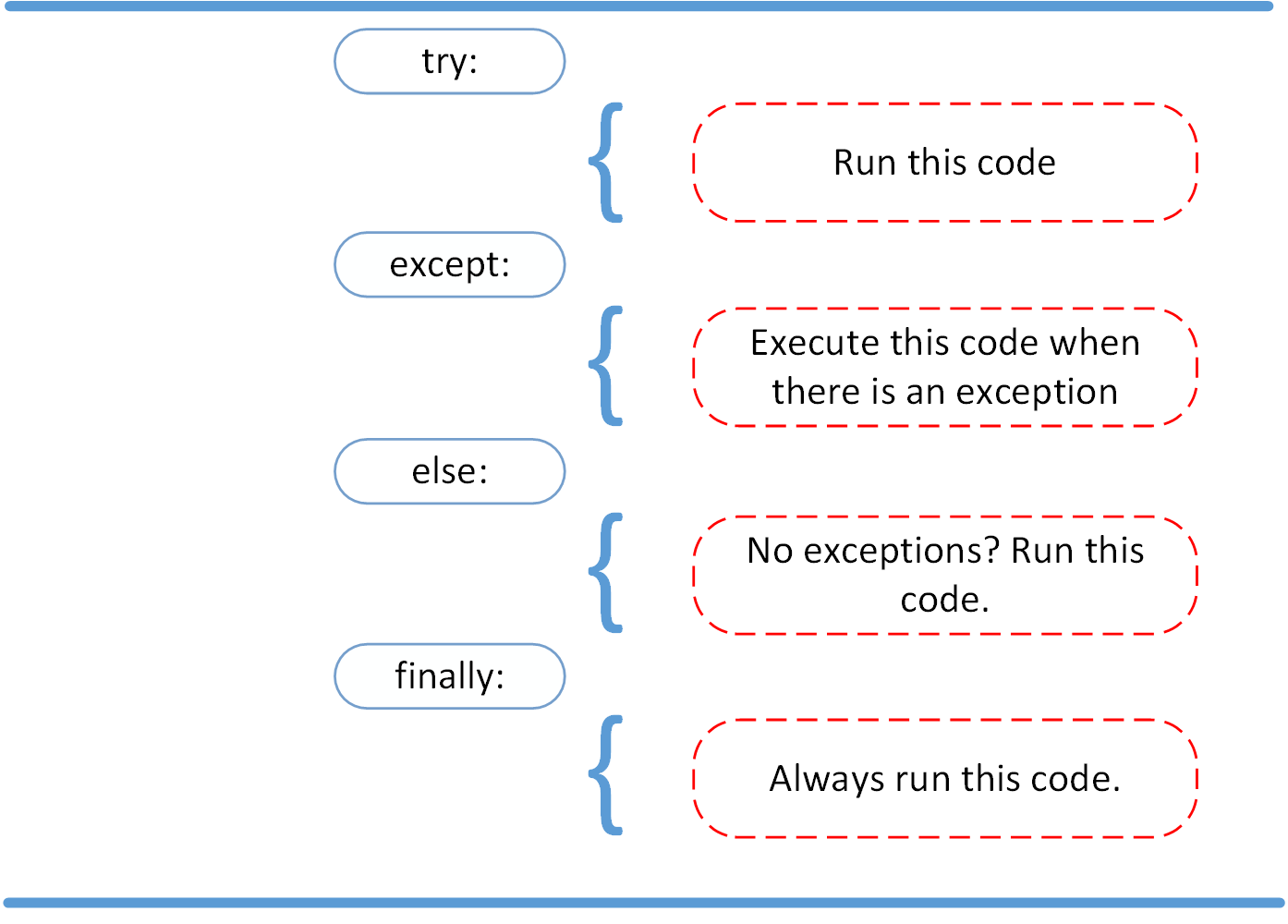


Figure 1: [Schematic from Website 1](https://www.datacamp.com/community/tutorials/exception-handling-python)[[2]](#footnote-2)

1. **Compares differences between Errors and Exceptions and describes common types of Python Errors. I found it especially memorable how it described errors: “Unconditionally fatal.”**
2. **References the standard exceptions hierarchy in the official Python documentation. I find the official documentation intimidating, so I’m glad less intimidating sources are exposing me to it.**

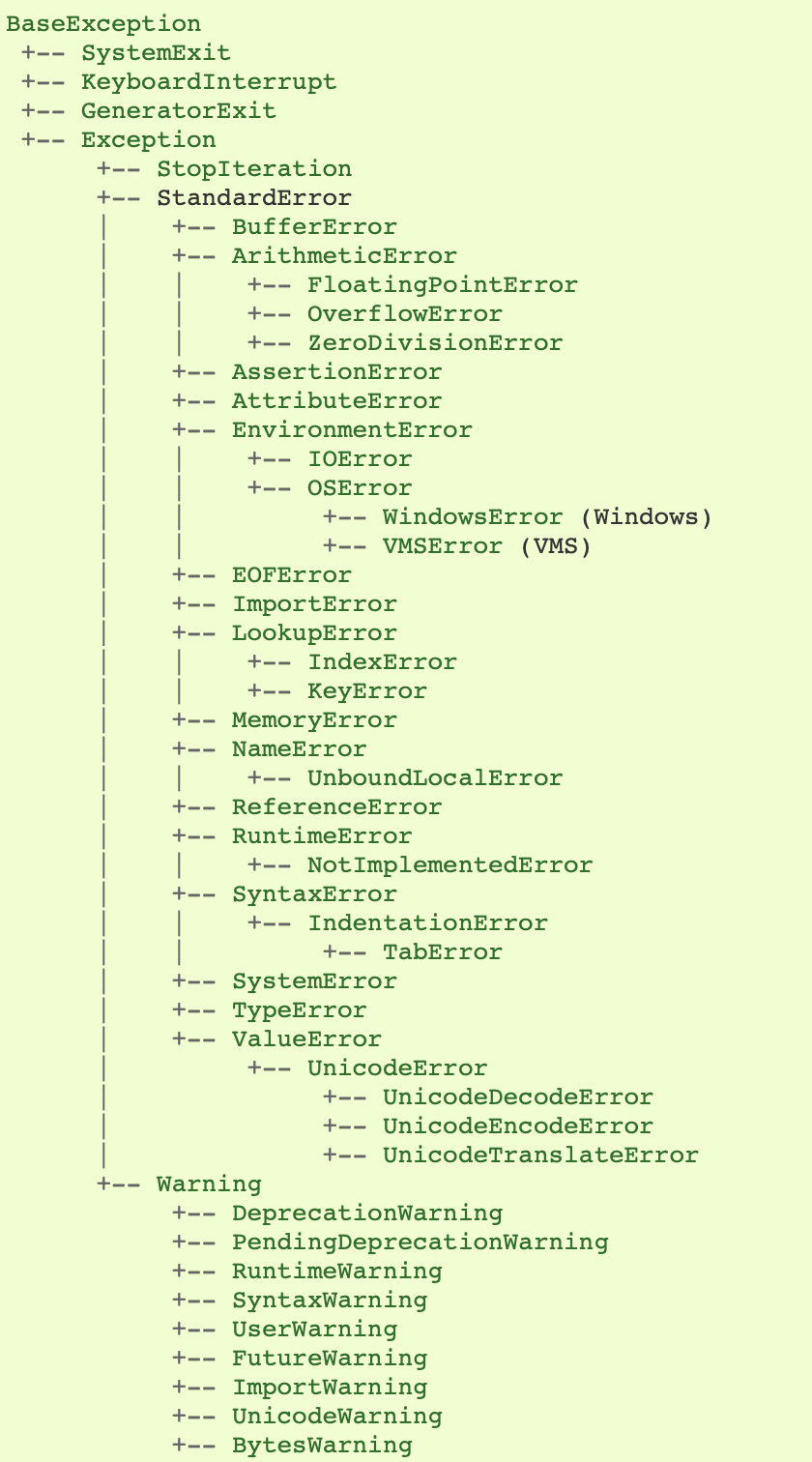
****

Figure 2 - [Exception hierarchy from Python documentation[[3]](#footnote-3)](https://docs.python.org/2/library/exceptions.html#exception-hierarchy)

1. **Explains disadvantages of using Exception Handling: slower running programs and increased code length.**

### **Website 2: “**[Exception Handling in Python” from Python for Beginners](https://www.pythonforbeginners.com/error-handling/exception-handling-in-python)[[4]](#footnote-4)

**Why is this website good?:**

**Provides an excellent description of how a sequential set of try/except/finally blocks flow into each other and, how the code relates, and what Python is doing when it reaches each section and/or encounters an error.**

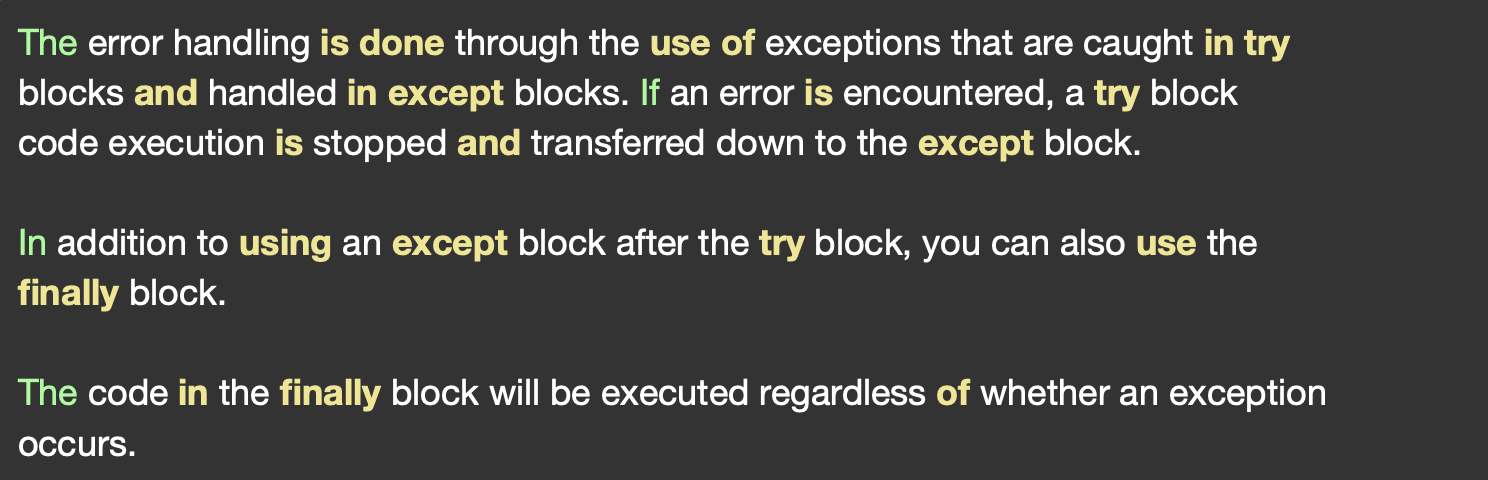
****

Figure 3 - ["How does it work" section from Website 2](https://www.pythonforbeginners.com/error-handling/exception-handling-in-python)[[5]](#footnote-5)

**Why is this website bad?:**

**Appears to be using syntax from Python 2.7, and it doesn’t mention this up front. This would confuse a beginner like me.**

# **Identifying websites that are good at describing Python’s pickling feature.**

## **Website 1: “**[Pickle in Python: Object Serialization](https://www.datacamp.com/community/tutorials/pickle-python-tutorial)” from Data Camp[[6]](#footnote-6)

Why is this website good?

1. Describes when not to use Pickling: I didn’t know that the pickling protocol was specific to Python and not interchangeable among programming languages. It also warns you that pickle files may contain malicious code that could executed when unpickled.
2. Describes Java Script Object Notation and compares it to Pickle. Highlights that JSON is language independent, more secure, and faster.
3. Describes how to unpickle something originally pickled in Python 2.
4. Describes how to compress pickle files.

## [Website 2: How to Pickle: “A Pickling and Unpicling Tutorial” from Python Central](https://www.pythoncentral.io/how-to-pickle-unpickle-tutorial/)[[7]](#footnote-7)

**Why is this website good?**

1. **Describes “data stream formats”, which are previous Pickling protocols associated with older versions of Python. Describes which formats are backwards compatible with previous versions of Python.**
2. **Describes Exceptions associated with the Pickle module.**
3. **Highlights that unreadability of the binary file provides a sort of data security (though not encrypted!).**

# **Modifying last week’s CD Inventory Script to 1) read/write data to binary files via the Pickle function and 2) integrate structured exception handling.**

## **Modifying script to read/write to binary files**

First, to use Pickle in Python, we must first import the Pickle module. I’ll do this using the syntax **import Pickle.[[8]](#footnote-8)**

We must change the name of the file we save to under the # DATA # section of the script. Let’s rename the extension on the file from .txt to .dat so that it is [clear we are saving data into a binary file.[[9]](#footnote-9)](https://youtu.be/nfDuGaTmuj8?t=246) So *strFileName = ‘CDInventory.txt’* changes to *strFileName = ‘CDInventory.dat’*. When we open the file later, if a **‘rb’** binary file access mode can’t find the file, it will throw an exception. For that reason, I’ll also go into the folder where the CDInventory.txt file resides and change the extension there to .dat as well.

**Next, we will modify all the functions defined under the class File Processor.**

**Let’s start with the function that reads the CD Inventory from a file and saves it volatile memory.**

1. **def** read\_file(file\_name, table):
2. table.clear()
3. objFile = open(file\_name, 'r')
4. **for** line **in** objFile:
5. data = line.strip().split(',')
6. dicRow = {'ID': int(data[0]), 'Title': data[1], 'Artist': data[2]}
7. table.append(dicRow)
8. objFile.close()

**In line 3 above, I will need to change the File Access mode from ‘r’ to ‘rb’ to indicate that we’re reading from a binary file.**[[10]](#footnote-10)

**When we unpickle a file containing a list, we’re essentially consuming it all at once, so we need to get rid of the for loop. We’ll replace it with the pickle.load() function**[[11]](#footnote-11) **and assign the result of the function to a list.**

**Our modifications result in:**

1. **def** read\_file(file\_name, table):
2. table.clear()  # this clears existing data and allows to load data from file
3. objFile = open(file\_name, 'rb')
4. table = pickle.load(objFile)
5. objFile.close()

**Moving onto the function to save the contents of the CD Inventory into a file.**

1. **def** write\_file(file\_name, table):
2. objFile = open(file\_name, 'w')
3. **for** row **in** table:
4. lstValues = list(row.values())
5. lstValues[0] = str(lstValues[0])
6. objFile.write(','.join(lstValues) + '\n')
7. objFile.close()

**We’ll use similar to logic as before, changing the write mode from ‘w’ to ‘wb’[[12]](#footnote-12) and getting rid of the for loop to write to the file. Instead of the write function we’ll use the pickle.dump()[[13]](#footnote-13) function.**

**Our modifications result in:**

1. **def** write\_file(file\_name, table):
2. objFile = open(file\_name, 'wb')
3. pickle.dump(table, objFile)
4. objFile.close()

## Integrating structured exception handling.

I envision two general types of exceptions that could arise in the program: exceptions thrown where users enter data of improper type and exceptions thrown when the script attempts to read a binary file that is not there.

Let’s first focus our efforts on the functions under class IO, where most of the user input takes place.

1. @staticmethod
2. **def** print\_menu():
4. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
5. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
7. @staticmethod
8. **def** menu\_choice():
9. choice = ' '
10. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
11. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
12. **print**()  # Add extra space for layout
13. **return** choice
15. @staticmethod
16. **def** show\_inventory(table):
17. **print**('======= The Current Inventory: =======')
18. **print**('ID\tCD Title (by: Artist)\n')
19. **for** row **in** table:
20. **print**('{}\t{} (by:{})'.format(\*row.values()))
21. **print**('======================================')

**def print\_menu()** doesn’t request user input, so there are no exceptions that can get thrown there. Nothing to do here.

**def menu\_choice** looks like a potential candidate for structured error handling since it requests user input. However, the function uses a clever **while** loop to limit the user the options that can be entered. I don’t think I’d be able to accomplish the same functionality using a **try / except** pair unless I created a custom exception, which I’m not allowed to use in this Assignment. Nothing to do here, either.

**def show\_inventory** doesn’t request user input, and apart from that, I don’t see anything that would possible throw and exception, so nothing to do here either.

1. @staticmethod
2. **def** input\_new\_cd():
3. cdID = input('Enter ID: ').strip()
4. title = input('What is the CD\'s title? ').strip()
5. artist = input('What is the Artist\'s name? ').strip()
6. **return** cdID, title, artist

Now, here at **def input\_new cd()**, we might be able to integrate some exception handling.

Nope, nevermind! Looks can be deceiving. Here there is nothing beyond accepting user input. But where is this data that is entered processed? It is processed at the **def** **add\_cd** function under class **Data Processor**:

1. @staticmethod
2. **def** add\_cd(cdID, title, artist, table):
3. intID = int(cdID)
4. row = {'ID': intID, 'Title': title, 'Artist': artist}
5. table.append(row)

Let’s add some exception handling here. To handle exceptions, you combine a **try** statement with an **except** clause.[[14]](#footnote-14) In the **try** statement, you put in the code that could be easily crash. Here we have possibility that the user didn’t enter a number when requested to enter a cd ID, say they entered “one” instead of “1”. If it’s not a number, the variable **cdID** can’t be converted to an integer. This would throw a **ValueError**, which is [“raised when an operation or function receives an argument that has the right type but an inappropriate value”](https://docs.python.org/3/library/exceptions.html).[[15]](#footnote-15)

Let’s arrange the try statement and except clause to handle this exception.

1. **try**: intID = int(cdID)
2. **except** ValueError:
3. **print**("You must enter the CD ID as an integer. Adding CD failed." )
4. **else**:
5. row = {'ID': intID, 'Title': title, 'Artist': artist}
6. table.append(row)

I ended up adding an **else** clause as well. This ensures that that code only executes if an exception is not raised.[[16]](#footnote-16) If an exception is raised no other code is executed in the function. The program goes back into the main **while** loop in the program and redisplays the menu, giving the user the opportunity to reenter the CD information. Alternatively, I could indent this code under the **try** block as well, with no change in functionality.

In retrospect, I probably could move the **int()** function from the **add\_cd()** function to the **input\_new\_cd()** function and convert the string to an integer there. That would allow me to run my exception handling earlier in the script. It would also allow me throw the **ValueError** exception immediately after the user entered the offending information and save the trouble of entering the remaining CD information.

The only other place where I could see user input causing the script to crash is in the main **while** loop, where the script seeks user input for CD to delete (line 3 below). Again, here the script converts the string entered into an integer using the **int()** function, and if the user does not enter a string as a number, the script will throw an exception, specifically another **ValueError**.

1. **elif** strChoice == 'd':
2. IO.show\_inventory(lstTbl)
3. intIDDel = int(input('Which ID would you like to delete? ').strip())
4. DataProcessor.delete\_cd(intIDDel, lstTbl)
5. IO.show\_inventory(lstTbl)
6. **continue**

Adding a **try** statement, **except** clause, and **else** clause gives us

1. **elif** strChoice == 'd':
2. IO.show\_inventory(lstTbl)
3. **try**: intIDDel = int(input('Which ID would you like to delete? ').strip())
4. **except** ValueError:
5. **print**("You must enter ID as a integer. Deleting CD failed.")
6. **else**:
7. DataProcessor.delete\_cd(intIDDel, lstTbl)
8. IO.show\_inventory(lstTbl)
9. **continue**

Again, I could probably move the code from **else** clause to under the **try** clause. Either way the code doesn’t execute unless a number is entered.

Now let’s shift focus to the error that can be raised when the program tries to read from a non existent binary file. This happens at def **read\_file()** under the **FileProcessor** class. The type of error that could be raised here is a IOError, which is [“raised when an I/O operation…fails for an I/O-related reason, e.g., ‘file not found’.”[[17]](#footnote-17)](https://docs.python.org/2/reference/simple_stmts.html#print)

1. **def** read\_file(file\_name, table):
2. table.clear()  # this clears existing data and allows to load data from file
3. objFile = open(file\_name, 'rb')
4. table = pickle.load(objFile)
5. objFile.close()

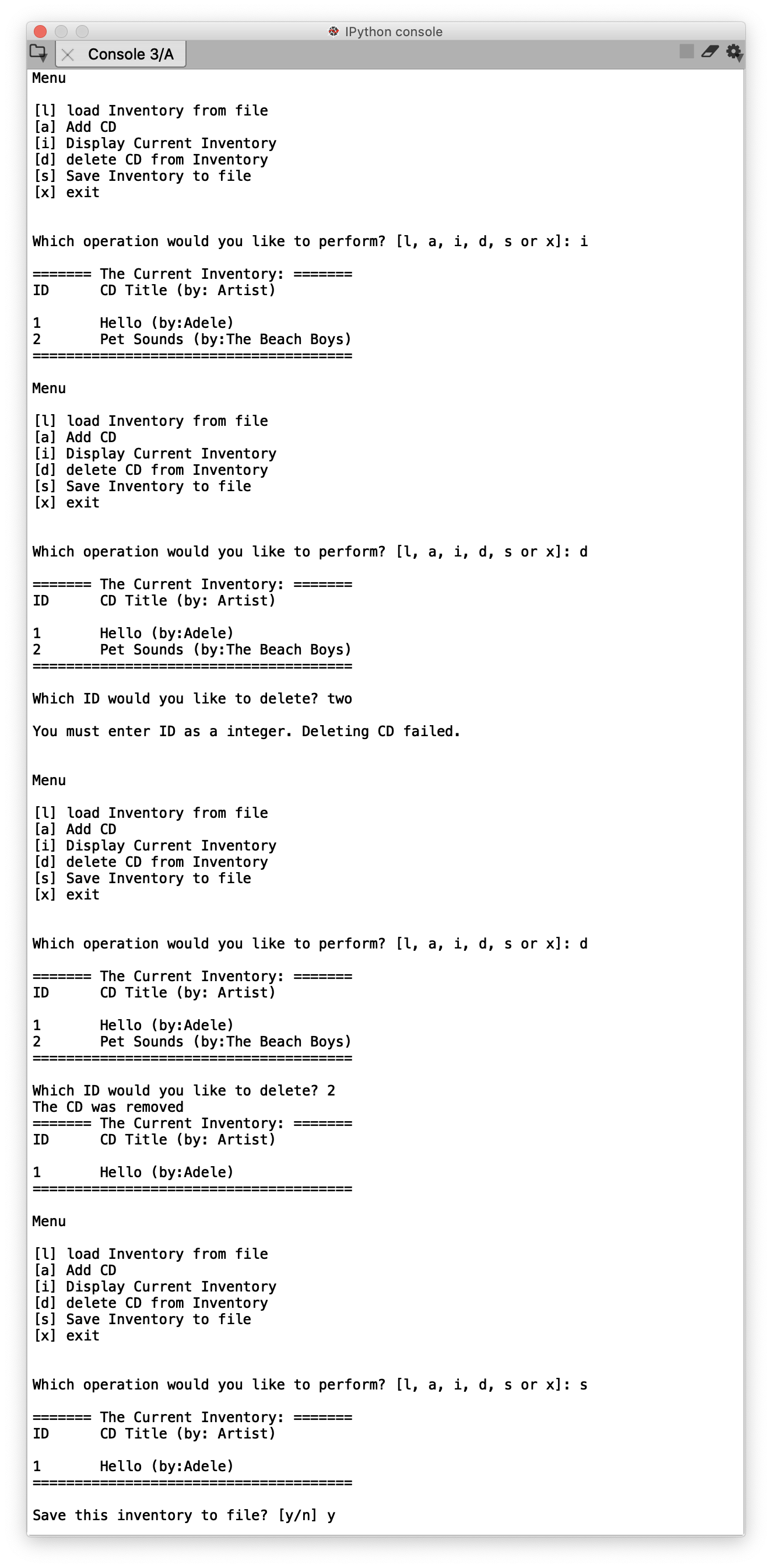
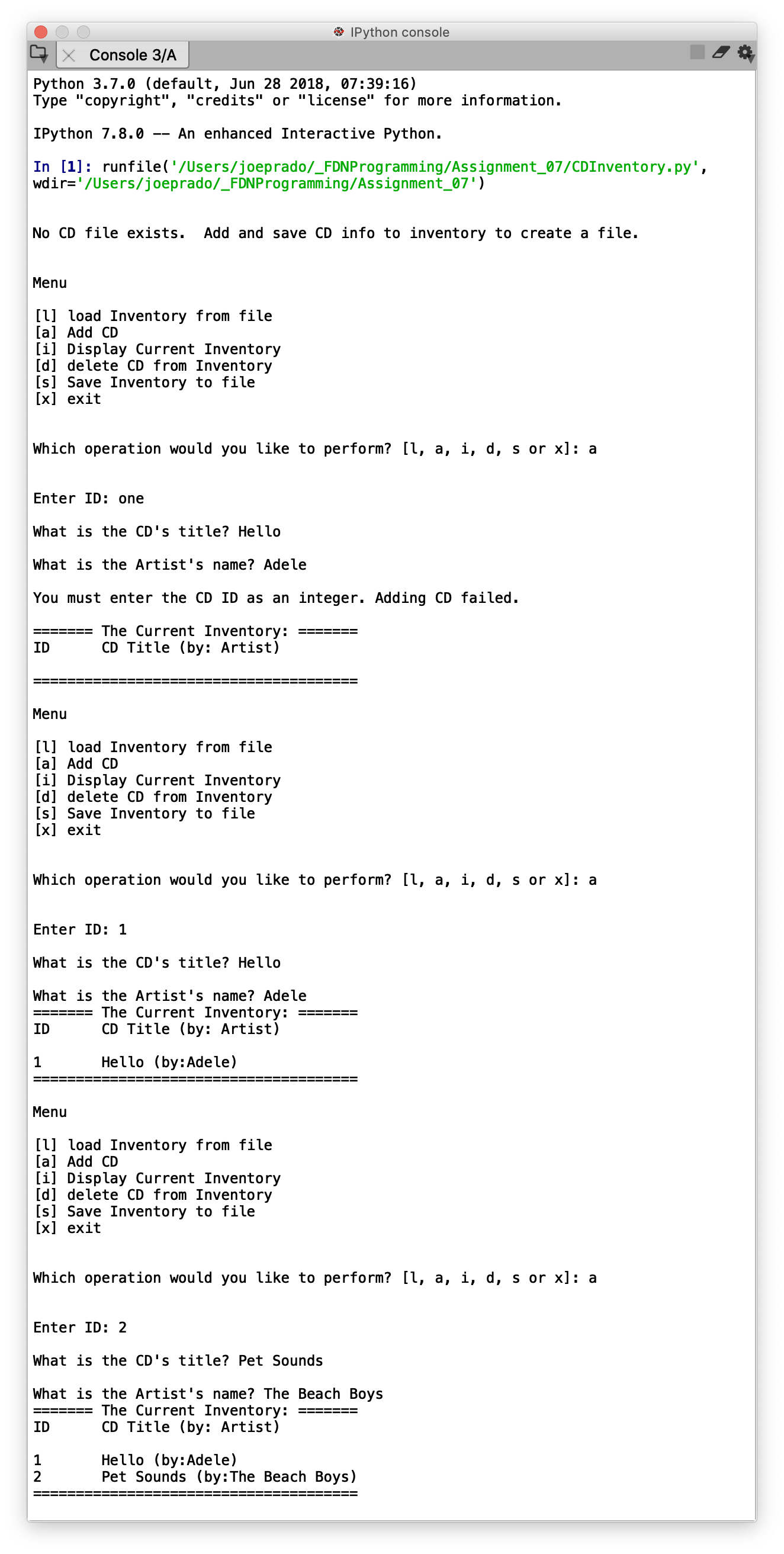
Adding our try statement ane except clause gives us:

1. **def** read\_file(file\_name, table):
2. table.clear()  # this clears existing data and allows to load data from file
3. **try**:
4. objFile = open(file\_name, 'rb')
5. table = pickle.load(objFile)
6. objFile.close()
7. **except** IOError:
8. **print**("No CD file exists.  Add and save CD info to inventory to create a file.")

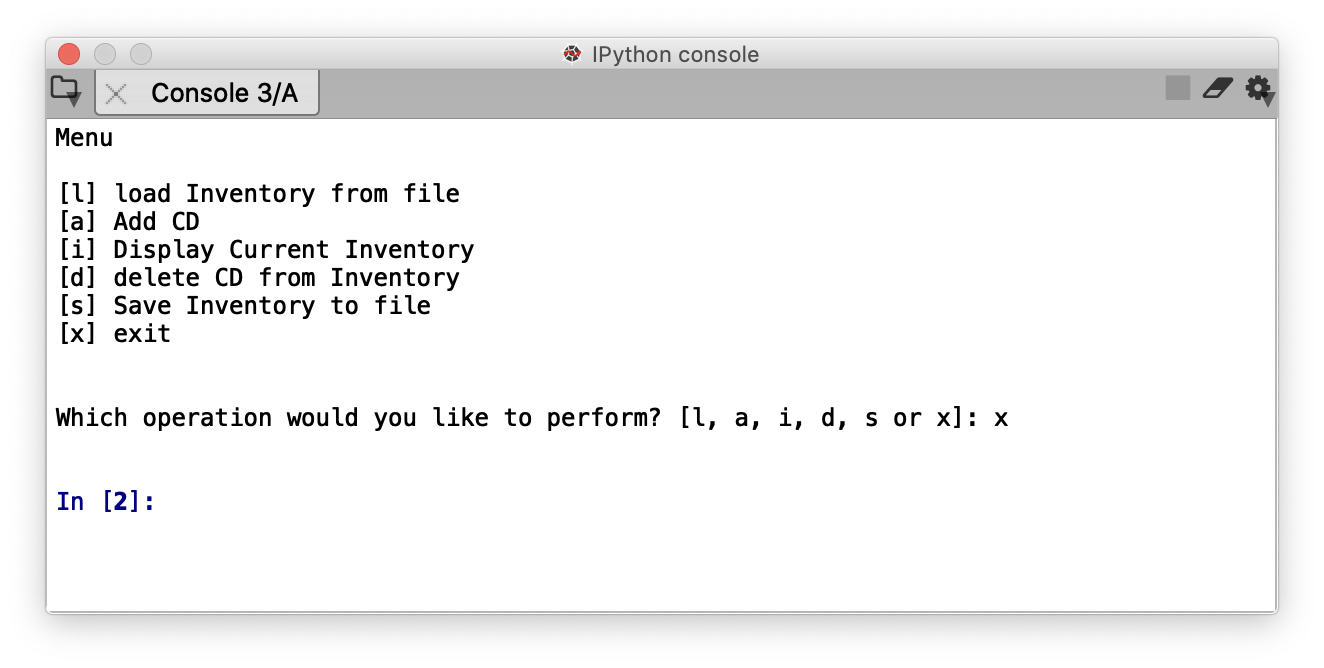
Fortunately, the ‘wb’ file access mode that is accessed later in the script will create a file if it doesn’t exist[[18]](#footnote-18), so I use the **print** function to simply leave instructions for the user to enter and save CD information.

# Running the script

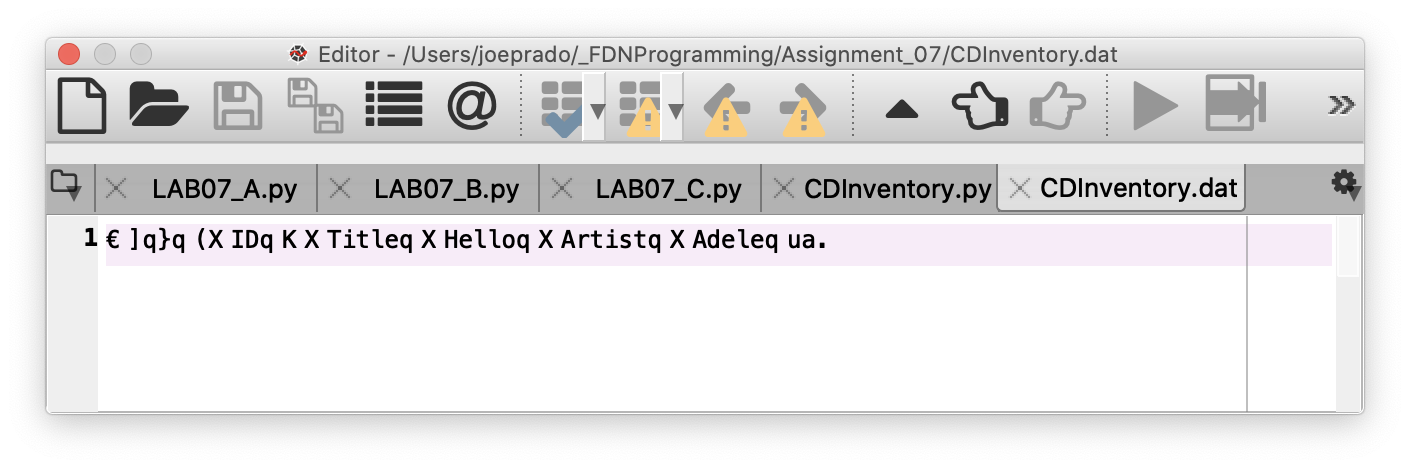
Running the script in Spyder results in the following:



*Figure 1 – Output from Spyder*



*Figure 1 – Output from Spyder (cont.)*

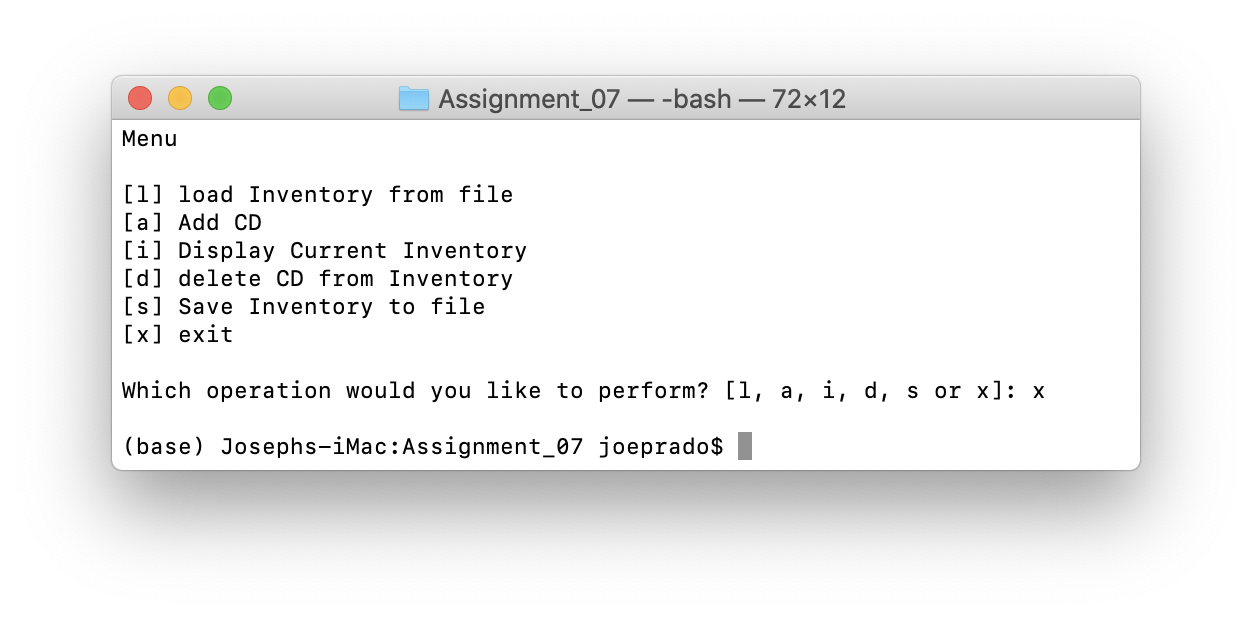
**

*Figure 2 – CDInventory.dat as written to by Spyder*

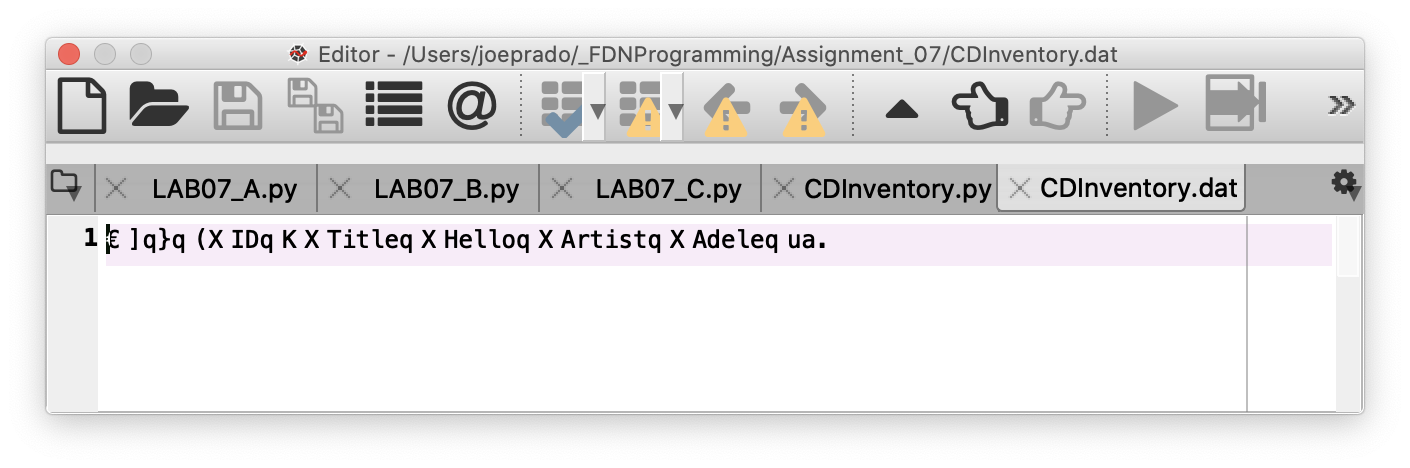
Running the script in the Terminal results in the following:

# 

# *Figure 3 – Output from Terminal*

**

# *Figure 3 – Output from Terminal (cont.)*

**

*Figure 3 – CDInventory.txt as written to by the Terminal.*

Link to GitHub

<https://github.com/joeprado/Assignment_07>

Summary

In this assignment, I:

1. Identified websites that are good at explaining Python’s Exception Handling.
2. Identified websites that are good at explaining Python’s pickling feature.
3. Modified an existing script to utilize structured exception handling around area where there is user interaction, type casting, or file access operations.
4. Modified an existing script to utilize the pickling feature to read and write from binary files.

# Appendix

## Listing CDInventory.py

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. # joeprado, 2020-Mar-09, Began modifying Script
7. # joeprado, 2020-Mar-10, Added docstrings.
8. # joeprado, 2020-Mar-10, Finished script modifications
9. # joeprado, 2020-Mar-23, Added structured exception handling
10. # joeprado, 2020-Mar-23, Modified script to use binary files instead of text files
11. # joeprado, 2020-Mar-24, Edited docstrings to reflect use of binary files
12. #------------------------------------------#
14. **import** pickle #Import Pickle module
16. # -- DATA -- #
17. strChoice = '' # User input
18. lstTbl = []  # list of lists to hold data
19. dicRow = {}  # list of data row
20. strFileName = 'CDInventory.dat'  # data storage file
21. objFile = None  # file object

24. # -- PROCESSING -- #
25. **class** DataProcessor:
26. @staticmethod
27. **def** add\_cd(cdID, title, artist, table):
28. """Adding user data for new CD to a table.
30. Takes user input fed into the function via parameters,formats it as a set key:value pairs in a
31. dictionary, and then appends that dictionary as row nested inside a list.
33. Args:
34. cdID (string): ID number of CD as entered by the user
35. title (string): Title of CD as entered by the user
36. artist (string): Artist name for the CD as entered by the user
37. table (list of dict): list of dictionaries that holds our data in volatile memory
39. Returns:
40. None.
41. """
42. **try**: intID = int(cdID)  #try statement, except clause, and else clause to prevent program from crashing if a numnber isn't entered.
43. **except** ValueError: # Handles Value Error if it is raised.
44. **print**("\nYou must enter the CD ID as an integer. Adding CD failed.\n" )
45. **else**:
46. row = {'ID': intID, 'Title': title, 'Artist': artist}
47. table.append(row)
49. @staticmethod
50. **def** delete\_cd(selectID, table):
51. """Delete a CD selected by user based on ID.
53. Takes user input for the ID number of a CD the user would like to delete, searches for the row (dictionary) the ID is in,
54. then deletes that row (dictionary) in the table (list).
56. Args:
57. selectID (integer): user selection for CD ID number the user would like to delete.
58. table (list of dict): list of dictionaries that holds our data in volatile memory
60. Returns:
61. None.
62. """
63. intRowNr = -1
64. blnCDRemoved = False
65. **for** row **in** table:
66. intRowNr += 1
67. **if** row['ID'] == selectID:
68. **del** table[intRowNr]
69. blnCDRemoved = True
70. **break**
71. **if** blnCDRemoved:
72. **print**('The CD was removed')
73. **else**:
74. **print**('Could not find this CD!')

77. **class** FileProcessor:
78. """Processing the data to and from text file"""
80. @staticmethod
81. **def** read\_file(file\_name, table):
82. """Function to load binary file to a list of dictionaries.
84. Loads data from file identified by file\_name into a 2D list

87. Args:
88. file\_name (string): name of pickled binary file used to load the data from
89. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
91. Returns:
92. None.
93. """
94. table.clear()  # this clears existing data and allows to load data from file
95. **try**:
96. objFile = open(file\_name, 'rb') #Reads from binary file in.
97. table = pickle.load(objFile) #Loads contents of binary file into list
98. objFile.close()
99. **except** IOError:
100. **print**("\nNo CD file exists.  Add and save CD info to inventory to create a file.\n")

103. @staticmethod
104. **def** write\_file(file\_name, table):
105. """Pickles the contents of CD Inventory into a binary file.
107. Takes the list of dictionaries identified by table and dumps it into binary file
108. identified by file\_name.

111. Args:
112. file\_name (string): name of file used to save data to.
113. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
115. Returns:
116. None
117. """
118. objFile = open(file\_name, 'wb') #Writes to binary file. Creates file if it doesn't exist.
119. pickle.dump(table, objFile) #Dumps content of list containing CD information into binary file.
120. objFile.close()

123. # -- PRESENTATION (Input/Output) -- #
125. **class** IO:
126. """Handling Input / Output"""
128. @staticmethod
129. **def** print\_menu():
130. """Displays a menu of choices to the user
132. Args:
133. None.
135. Returns:
136. None.
137. """
139. **print**('\nMenu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
140. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
142. @staticmethod
143. **def** menu\_choice():
144. """Gets user input for menu selection
146. Args:
147. None.
149. Returns:
150. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
152. """
153. choice = ' '
154. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
155. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
156. **print**()  # Add extra space for layout
157. **return** choice
159. @staticmethod
160. **def** show\_inventory(table):
161. """Displays current inventory table

164. Args:
165. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
167. Returns:
168. None.
169. """
170. **print**('======= The Current Inventory: =======')
171. **print**('ID\tCD Title (by: Artist)\n')
172. **for** row **in** table:
173. **print**('{}\t{} (by:{})'.format(\*row.values()))
174. **print**('======================================')
176. @staticmethod
177. **def** input\_new\_cd():
178. """Function that collects user input for a new CD to be added to inventory.
180. Args:
181. None.
183. Returns:
184. cdID (string): string representing ID number user entered for CD
185. title (string): string representing CD title entered by user
186. artist (string): string representing artist name entered by user
187. """
188. cdID = input('Enter ID: ').strip()
189. title = input('What is the CD\'s title? ').strip()
190. artist = input('What is the Artist\'s name? ').strip()
191. **return** cdID, title, artist

194. # 1. When program starts, calls function that reads in the currently saved Inventory
195. **print**()
196. FileProcessor.read\_file(strFileName, lstTbl)
198. # 2. start main loop
199. **while** True:
200. # 2.1 Display Menu to user and get choice
201. IO.print\_menu()
202. strChoice = IO.menu\_choice()
204. # 3. Process menu selection
205. # 3.1 process exit first
206. **if** strChoice == 'x':
207. **break**
208. # 3.2 process load inventory
209. **if** strChoice == 'l':
210. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
211. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
212. **if** strYesNo.lower() == 'yes':
213. **print**('reloading...')
214. FileProcessor.read\_file(strFileName, lstTbl) #Calls function that loads text file containing CD inventory into runtime.
215. IO.show\_inventory(lstTbl) #Calls function that displays inventory to user
216. **else**:
217. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
218. IO.show\_inventory(lstTbl) #Calls function that displays inventory to user
219. **continue**  # start loop back at top.
220. # 3.3 process add a CD
221. **elif** strChoice == 'a':
222. # 3.3.1 Calls function that asks user for new ID, CD Title and Artist
223. strID, strTitle, strArtist = IO.input\_new\_cd()
224. # 3.3.2 Calls the function that adds item to the table
225. DataProcessor.add\_cd(strID, strTitle, strArtist, lstTbl)
226. #Calls function that displays inventory with added CD
227. IO.show\_inventory(lstTbl)
228. **continue**  # start loop back at top.
229. # 3.4 process display current inventory
230. **elif** strChoice == 'i':
231. IO.show\_inventory(lstTbl) # Calls function that displays current inventory
232. **continue**  # start loop back at top.
233. # 3.5 process delete a CD
234. **elif** strChoice == 'd':
235. # 3.5.1 get Userinput for which CD to delete
236. # 3.5.1.1 Calls function that displays inventory to user
237. IO.show\_inventory(lstTbl)
238. # 3.5.1.2 ask user which ID to remove
239. **try**: intIDDel = int(input('Which ID would you like to delete? ').strip()) # try statement to prevent prgm crash if non-number is entered.
240. **except** ValueError: # handles ValueError exception and notified user that deleting CD failed.
241. **print**("\nYou must enter ID as a integer. Deleting CD failed.\n")
242. # 3.5.2 Calls function that searches thru table and deletes CD
243. **else**:
244. DataProcessor.delete\_cd(intIDDel, lstTbl)
245. IO.show\_inventory(lstTbl) #Calls function that displays inventory to user
246. **continue**  # start loop back at top.
247. # 3.6 process save inventory to file
248. **elif** strChoice == 's':
249. # 3.6.1 Calls function that displays current inventory.
250. IO.show\_inventory(lstTbl)
251. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower() #asks user for confirmation to save
252. # 3.6.2 Process choice
253. **if** strYesNo == 'y':
254. # 3.6.2.1 Calls function that saves data
255. FileProcessor.write\_file(strFileName, lstTbl)
256. **else**:
257. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
258. **continue**  # start loop back at top.
259. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be save:
260. **else**:
261. **print**('General Error')

1. Accessed 23 Mar 2020 [↑](#footnote-ref-1)
2. Accessed 23 Mar 2020 [↑](#footnote-ref-2)
3. Accessed 23 Mar 2020 [↑](#footnote-ref-3)
4. Accessed 23 Mar 2020 [↑](#footnote-ref-4)
5. Accessed 23 Mar 2020 [↑](#footnote-ref-5)
6. Accessed 23 March 2020 [↑](#footnote-ref-6)
7. Accessed 23 March 2020 [↑](#footnote-ref-7)
8. Page 200, Python Programming for Absolute Beginner, Third Edition [↑](#footnote-ref-8)
9. Accessed 23 March 2020 [↑](#footnote-ref-9)
10. Page 201, Python Programming for the absolute beginner, Third Edition. [↑](#footnote-ref-10)
11. Page 203, Python Programming for the absolute beginner, Third Edition. [↑](#footnote-ref-11)
12. Page 201, Python Programming for the absolute beginner, Third Edition. [↑](#footnote-ref-12)
13. Page 203, Python Programming for the absolute beginner, Third Edition. [↑](#footnote-ref-13)
14. Page 206, Python Programming for the absolute beginner, Third Edition. [↑](#footnote-ref-14)
15. Accessed 24 Mar 2020 [↑](#footnote-ref-15)
16. Page 210, Python Programming for the absolute beginner, Third Edition. [↑](#footnote-ref-16)
17. Accessed 23 March 2020 [↑](#footnote-ref-17)
18. Page 201, Python Programming for the absolute beginner, Third Edition. [↑](#footnote-ref-18)