Christopher J. Scott

24-Feb-2021

IT FDN 110: Introduction to Programming (Python)

Assignment 06

Creating and calling upon functions, classes and docstrings within Python

# Introduction

Module 06 introduced functions and highlighted their power when it comes to consolidation and avoiding repetitiveness within a program. Separation of concerns and classes were also expanded on in order to further improve formatting of larger programs. This module culminated with modifying an existing code to incorporate classes and functions which resulted in a fairly streamlined main body of the program.

# Basics of functions

In order to utilize functions, they must first be defined inside the program before being executed by typing def followed by the name of the function. The subsequent indented lines are the contents of the function and are executed by being called upon later in the program. A simple function illustrated below [print\_menu()] contains only 2 lines of functioning content (lines 126 & 127) and will simply print the strings shown whenever called upon. This is helpful within the CDInventory.py program as the menu of choices is something the user needs to reference multiple times as they use and interact with the program.

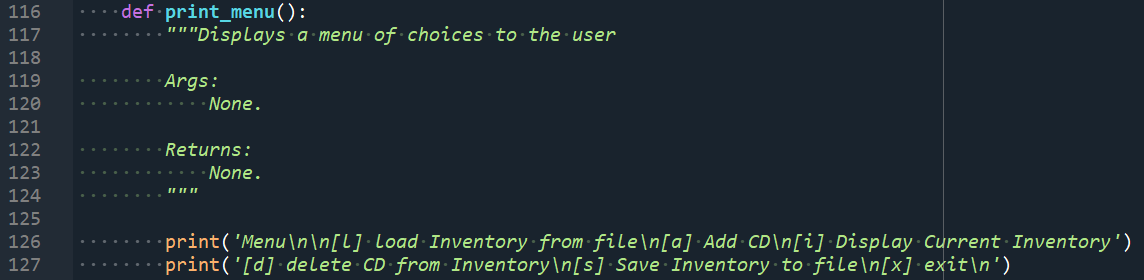


Figure . Simple function print\_menu()

While the function shown above does not have any arguments, functions have the ability to pass through parameters or arguments and those arguments can be referenced or used within the function. It is important to note that the exact name of the argument as defined in the function is only relevant within the function definition. When called upon, the argument names may change while the function will still associate the individual arguments based on the order they were included after the name of the function. An example of a function definition that passes in arguments is shown below as read\_file(file\_name, table).

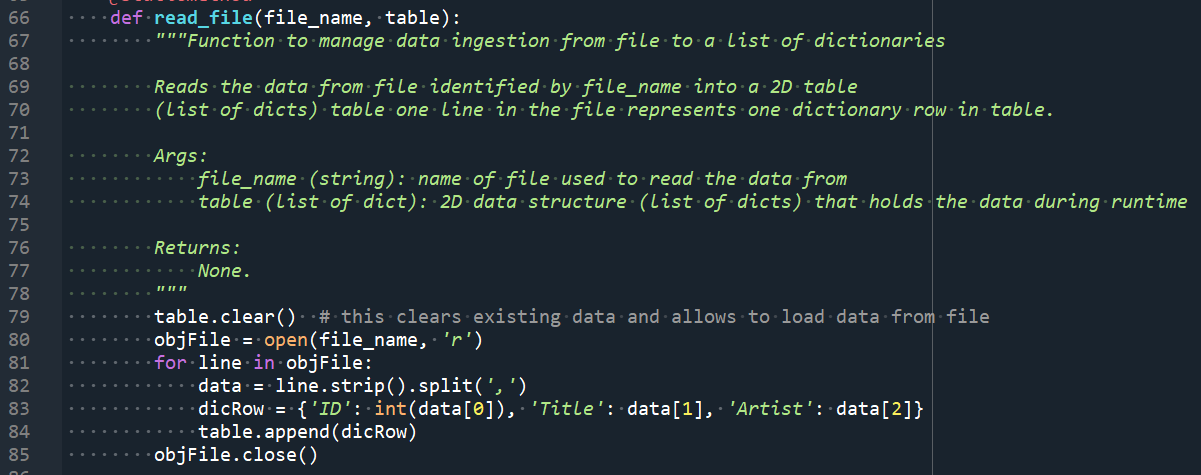


Figure . read\_file() function with 2 arguments included

In order to call functions within the main body of a program, the name of the function is simply called and arguments are included if necessary. The power of functions are highlighted below in lines 224 – 230. Within the 6 line sequence, 3 separate functions are called which each execute a particular block of code to display the inventory (show\_inventory()), delete a particular CD (delete\_CD()) and display the inventory again. Without functions, this block of code would have been significantly longer and had instances of duplication.

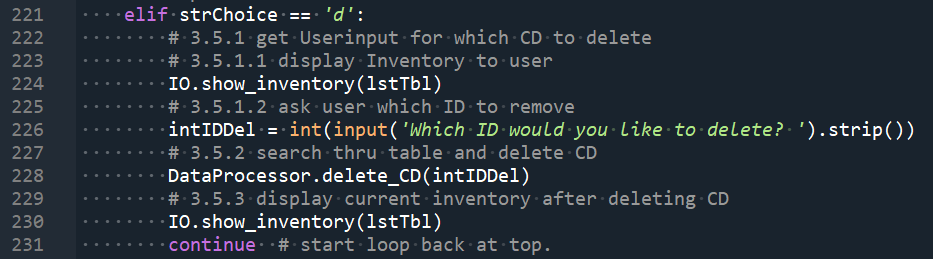


Figure . Main body code calling on functions

# Formatting using classes and docstrings

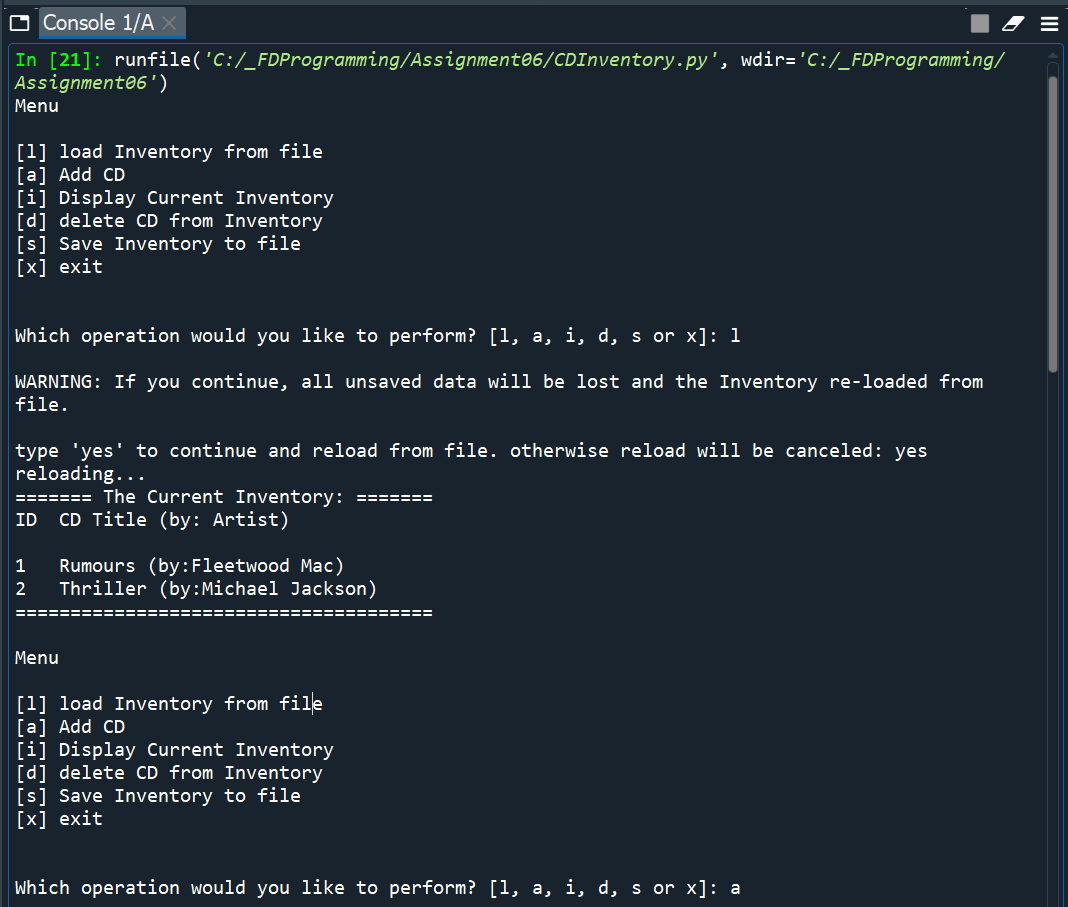
As was demonstrated in Module 05, separation of concerns is a useful strategy to format large codes into smaller subsets based on their functionality. As was previously demonstrated, three distinct sections (data, processing and presentation) are common categories that specific functions can fall within. For Module 06, the classes used are DataProcessor for processing data within the current inventory, FileProcessor for processing data to and from a .txt file and IO which handles all the input/output from the user of the program. When functions are put in specific classes, it is important to note that in order to call the function, the main body must call Class.Function\_Name(arg) as shown in Figure 3.

An additional feature that is introduced to improve program formatting is *docstrings* which provide additional information to anyone reading the code or calling the function. The details of what the function accomplishes, argument type and details and return type and details are provided to assist users with ensuring the functions are called and executed correctly.

# Reworking and validating the functionality of CDInventory.py

The objective of Assignment 6 was to rework an existing partially complete by rearranging lines into functions and classes and create new functions to fill some functional gaps. Two DataProcessor class functions were created to add and delete a particular CD. An additional IO class function was created that allowed for input of specific CD ID, title and artist information when adding a new CD to the inventory. Finally, some syntax was changed to allow for the inventory data to be stored as a list of dictionaries rather than a list of lists.

To demonstrate the functionality of the CDInventory.py program, an initial CDInventory.txt file was created and saved to the same local directory as the CDInventory.py program. When executed in Spyder, Figure 4 shows the resulting output after a user interaction.



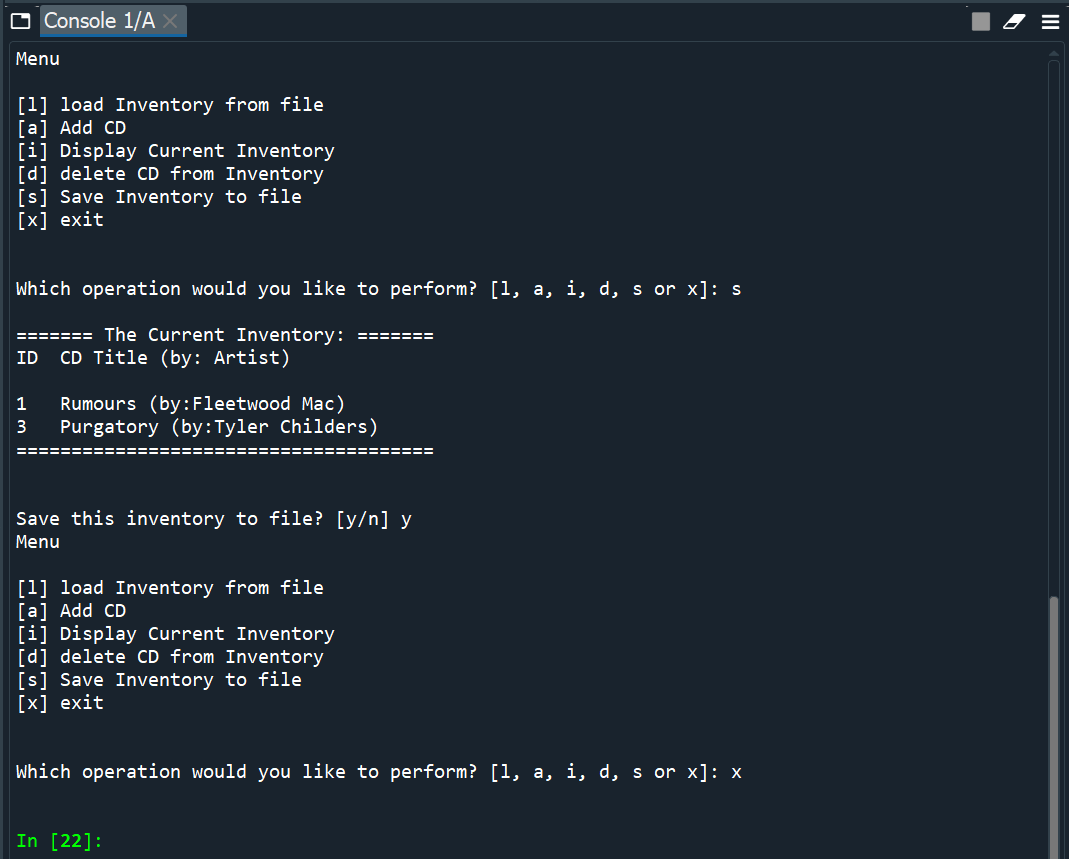
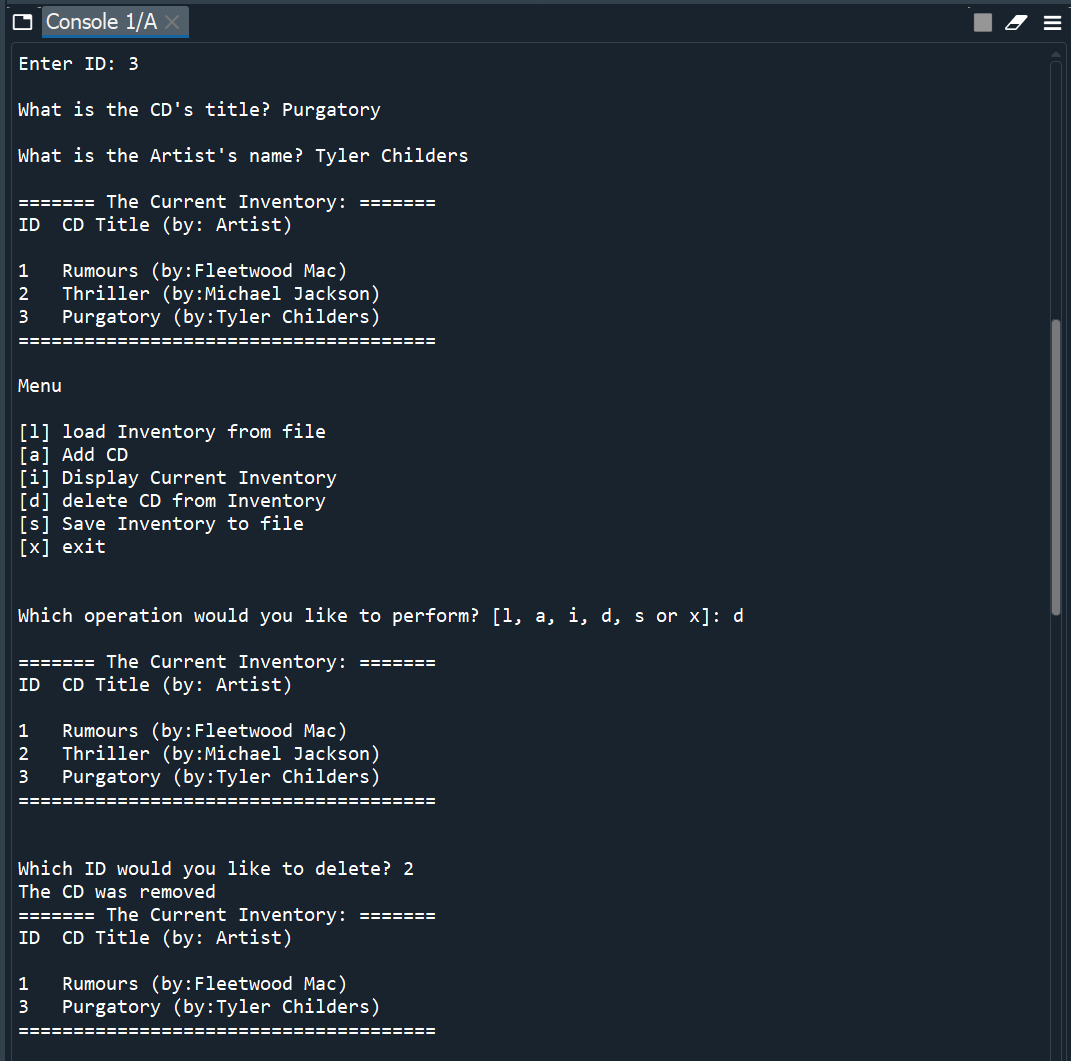


Figure . Result of running CDInventory.py within Spyder

Upon saving the inventory to file, the CDInventory.txt file results in Figure 5.

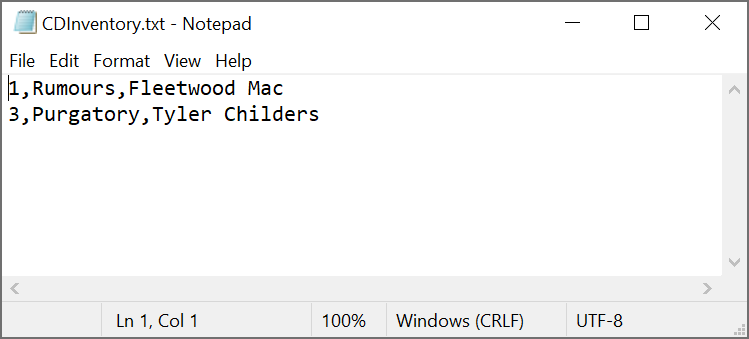


Figure . Result of saving within CDInventory.py

A brief demonstration of the CDInventory.py programming running in terminal is shown in Figure 6 in the Appendix.

# Summary

By completing Module 6 and associated Lab exercises, a user is able to create a program that creates and calls functions that are grouped into particular classes highlighting the benefits of separation of concerns and docstrings. As was the case in Module 05, a repository that contains this specific CDInventory.py for Module 06 can be found at <https://github.com/cscott08vt/Assignment06> .

# Appendix

1. #------------------------------------------#
2. # Title: Assignment06.py
3. # Desc: Working with classes and functions.
4. # Change Log: (Who, When, What)
5. # CScott, 2021-Feb-23, Created File
6. # CScott, 2021-Feb-24, Revised DataProcessor functions
7. #------------------------------------------#
9. # -- DATA -- #
10. strChoice = '' # User input
11. lstTbl = []  # list of dictionaries to hold data
12. dicRow = {}  # dictionary of data row
13. strFileName = 'CDInventory.txt'  # data storage file
14. objFile = None  # file object
15. CD\_info = [None, None, None] # CD ID, title and artist
17. # -- PROCESSING -- #
18. **class** DataProcessor:
19. """Processing the data within the current inventory"""
21. @staticmethod
22. **def** add\_CD(CD\_info):
23. """Adds CD to current inventory
25. Args:
26. CD\_info (list): list of strgings containing CD ID, title and artist
28. Returns:
29. None
30. """
31. intID = int(CD\_info[0]) # Extracts intID from list CD\_info
32. strTitle = str(CD\_info[1]) # Extracts strTitle from list CD\_info
33. strArtist = str(CD\_info[2]) # Extracts strArtist from list CD\_info
34. dicRow = {'ID': intID, 'Title': strTitle, 'Artist': strArtist} # Creates dictionary containing all CD info
35. lstTbl.append(dicRow)
37. @staticmethod
38. **def** delete\_CD(intIDDel):
39. """Deletes a specific CD from current inventory based on ID
41. Args:
42. intIDDel (integer): ID of the CD to be deleted from current inventory
44. Returns:
45. None.
46. """
47. intRowNr = -1
48. blnCDRemoved = False
49. **for** row **in** lstTbl:
50. intRowNr += 1
51. **if** row['ID'] == intIDDel:
52. **del** lstTbl[intRowNr]
53. blnCDRemoved = True
54. **break**
55. **if** blnCDRemoved:
56. **print**('The CD was removed')
57. **else**:
58. **print**('Could not find this CD!')
59. **pass**

62. **class** FileProcessor:
63. """Processing the data to and from text file"""
65. @staticmethod
66. **def** read\_file(file\_name, table):
67. """Function to manage data ingestion from file to a list of dictionaries
69. Reads the data from file identified by file\_name into a 2D table
70. (list of dicts) table one line in the file represents one dictionary row in table.
72. Args:
73. file\_name (string): name of file used to read the data from
74. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
76. Returns:
77. None.
78. """
79. table.clear()  # this clears existing data and allows to load data from file
80. objFile = open(file\_name, 'r')
81. **for** line **in** objFile:
82. data = line.strip().split(',')
83. dicRow = {'ID': int(data[0]), 'Title': data[1], 'Artist': data[2]}
84. table.append(dicRow)
85. objFile.close()
87. @staticmethod
88. **def** write\_file(file\_name, table):
89. """Function to manage exporting data from list of dictionaries to a file
91. Writes the data to file identified by file\_name from a 2D table
92. (list of dicts) table one line in the file represents one dictionary row in table.
94. Args:
95. file\_name (string): name of file that the data is written to
96. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
98. Returns:
99. None.
100. """
101. objFile = open(file\_name, 'w')
102. **for** row **in** table:
103. lstValues = list(row.values())
104. lstValues[0] = str(lstValues[0])
105. objFile.write(','.join(lstValues) + '\n')
106. objFile.close()
107. **pass**

110. # -- PRESENTATION (Input/Output) -- #
112. **class** IO:
113. """Handling Input / Output"""
115. @staticmethod
116. **def** print\_menu():
117. """Displays a menu of choices to the user
119. Args:
120. None.
122. Returns:
123. None.
124. """
126. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
127. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
129. @staticmethod
130. **def** menu\_choice():
131. """Gets user input for menu selection
133. Args:
134. None.
136. Returns:
137. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
139. """
140. choice = ' '
141. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
142. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
143. **print**()  # Add extra space for layout
144. **return** choice
146. @staticmethod
147. **def** show\_inventory(table):
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 of strings): list of strings containing CD ID, title and artist
174. """
175. strID = input('Enter ID: ').strip()
176. strTitle = input('What is the CD\'s title? ').strip()
177. strArtist = input('What is the Artist\'s name? ').strip()
178. **print**() # Add extra space for layout
179. CD\_info = [strID, strTitle, strArtist]
180. **return** CD\_info
182. # 1. When program starts, read in the currently saved Inventory
183. FileProcessor.read\_file(strFileName, lstTbl)
185. # 2. start main loop
186. **while** True:
187. # 2.1 Display Menu to user and get choice
188. IO.print\_menu()
189. strChoice = IO.menu\_choice()
191. # 3. Process menu selection
192. # 3.1 process exit first
193. **if** strChoice == 'x':
194. **break**
195. # 3.2 process load inventory
196. **if** strChoice == 'l':
197. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
198. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
199. **if** strYesNo.lower() == 'yes':
200. **print**('reloading...')
201. FileProcessor.read\_file(strFileName, lstTbl)
202. IO.show\_inventory(lstTbl)
203. **else**:
204. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
205. IO.show\_inventory(lstTbl)
206. **continue**  # start loop back at top.
207. # 3.3 process add a CD
208. **elif** strChoice == 'a':
209. # 3.3.1 Ask user for new ID, CD Title and Artist
210. CD\_info=IO.ID\_title\_artist\_add()
211. # 3.3.2 Add item to the table
212. DataProcessor.add\_CD(CD\_info)
213. # 3.3.3 Display current inventory after adding CD
214. IO.show\_inventory(lstTbl)
215. **continue**  # start loop back at top.
216. # 3.4 process display current inventory
217. **elif** strChoice == 'i':
218. IO.show\_inventory(lstTbl)
219. **continue**  # start loop back at top.
220. # 3.5 process delete a CD
221. **elif** strChoice == 'd':
222. # 3.5.1 get Userinput for which CD to delete
223. # 3.5.1.1 display Inventory to user
224. IO.show\_inventory(lstTbl)
225. # 3.5.1.2 ask user which ID to remove
226. intIDDel = int(input('Which ID would you like to delete? ').strip())
227. # 3.5.2 search thru table and delete CD
228. DataProcessor.delete\_CD(intIDDel)
229. # 3.5.3 display current inventory after deleting CD
230. IO.show\_inventory(lstTbl)
231. **continue**  # start loop back at top.
232. # 3.6 process save inventory to file
233. **elif** strChoice == 's':
234. # 3.6.1 Display current inventory and ask user for confirmation to save
235. IO.show\_inventory(lstTbl)
236. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
237. # 3.6.2 Process choice
238. **if** strYesNo == 'y':
239. # 3.6.2.1 save data
240. FileProcessor.write\_file(strFileName, lstTbl)
241. **else**:
242. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
243. **continue**  # start loop back at top.
244. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be safe:
245. **else**:
246. **print**('General Error')

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

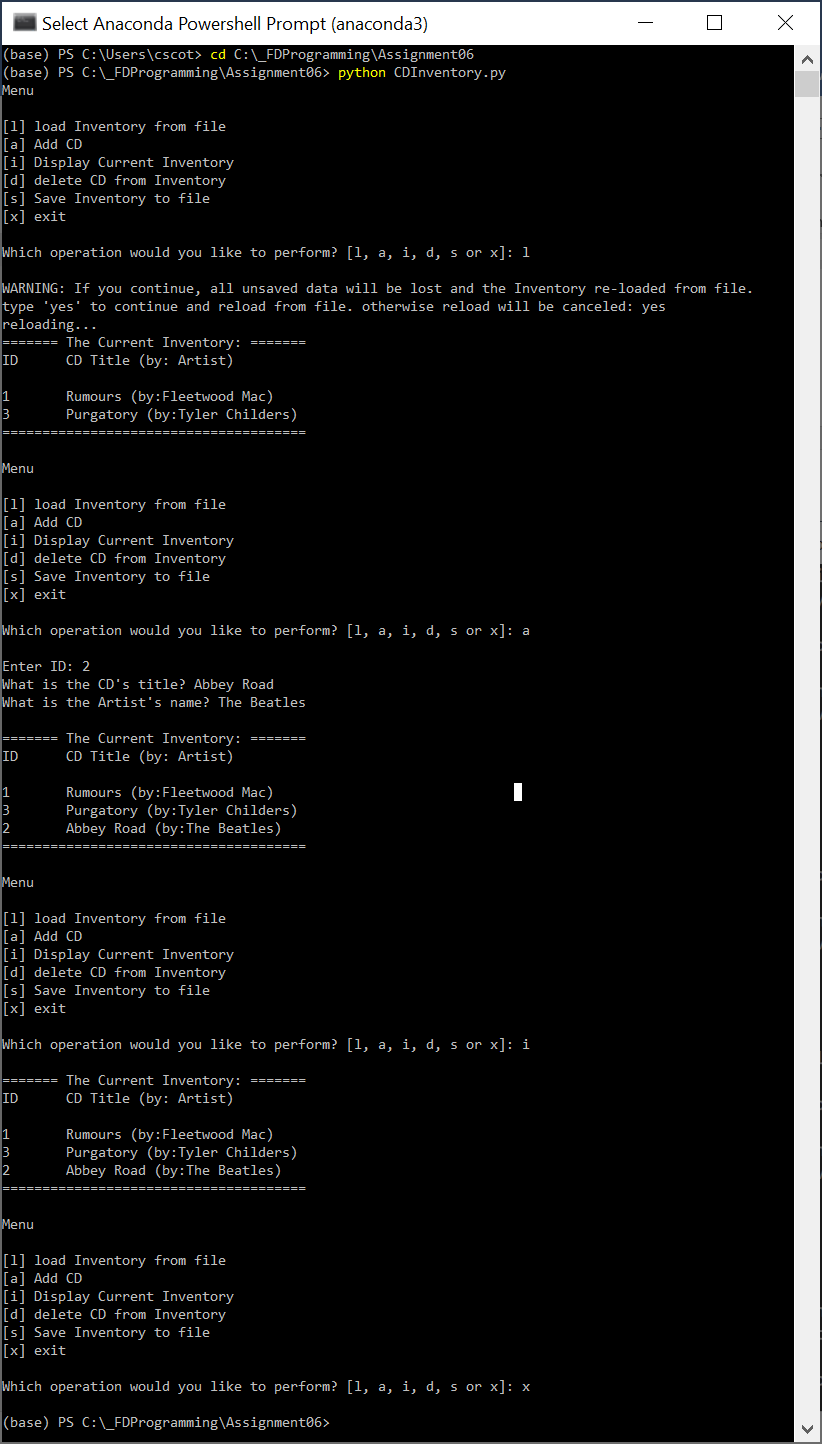


Figure . CDInventory.py being run in terminal