John Hunter

March 16, 2020

Foundations of Programing: Python

Module 08

Knowledge Document

Introduction

The primary discussion topic for this module is Object Oriented Programming(OOP). An introduction to this large topic is covered as an extension of many of the ancillary topics the class has covered previously. The definition of a class, and the use of objects have been large parts of the material from modules 06 and 07. Here though, the class is used to define an object, which has not previously been the principle subject. Using the definition of the object can be accomplished in myriad fashions, specifically the module applies the class and object definitions to the CDInventory application that has been the subject of many modules. A brief discussion of access controls and their use in scripted applications is also offered.

Materials Discussion

Classes can be used to define objects. The rudimentary class definitions used in earlier assignments provided a simple example where the class essentially contained only the functions needed to perform the tasks of the CDInventory application. However, the data structure that enabled the internal function was defined outside the class. Class definitions often package the definition of the data with the functionality. The object is then an instantiation of the class definition. Each instance of the class can be customized to create unique objects within the application. The instances of the objects remain sequestered, inheriting only from the definition. There is no propagation once data is updated in one object to a separate instance, unless the developer specifically intends and write that propagation into a custom inheritance from one object to another.

Python has a base class called ‘Object’ that will act as the default class from which the definition will inherit if no parent class is defined. Fields, which act similar to how variables have been used up to this point in the course, can be used to further define the class. The value that any field takes on can be referred to with syntax that is similar to the reference without the class definition, e.g. “objectName.fieldName”[1].

Constructors are methods that serve to properly type the data element of a class, at the point when an object is created from the class definition. Constructors can remove the ambiguity of untyped data. The Python implementation of the constructor method is the dunder init, ‘\_\_init\_\_()’. The call of this dunder is implied when the class name is used like a function call, e.g. objCan1 = CanOnAString('There is no spoon')[1]. Conversely, the destructor methods remove value definition of the targeted object, depending on the intent of the developer. The use of destructors is relied upon in some languages or applications when memory concerns dominate. The topic of destructors is beyond the scope of the material, but are mentioned for awareness.

The unofficial keyword ‘self’ is a convention in Python that is used often. It is implied with every method used in Python as a reflexive argument pointing back to the object itself. Python can then target the method to the object that the user wishes to manipulate.

Attributes can be defined at either the class level or at the instance level[3]. Defining an attribute at the class level allows all instances to share the definition of the attribute. This is desirable on some occasions. If this type of inheritance is not

Properties can be used on attributes to control the manipulation of the value. This set of methods represents one way that the user can control the validity of the values. The methods generally come in two flavors, getting the attribute, and setting the attribute. They are sometimes referred to by these terms, “getter” and “setter”[1]. When an attribute is defined, it can be set to private by adding a double underscore to the start of its name, e.g. ”self.\_\_message”[1]. A getter method will be required for any attribute set to private, and setter attributes can then provide a measure of validation for the value as it is assigned. Decorators are used to define the getter, which must be defined first, and the setter.

As a matter of practice in OOP, data in a class is only manipulated through methods and properties. This restriction allows for the class itself to remain a black box to an external interaction. The private/method, setter/getter layer of abstraction is not as robust as a formal data structure written into an application, and can be violated if the user so desires[1].

Type hints can be used to elaborate on the data type given and data type expected through a given attribute. This can be very useful given the loose restrictions on data typing on attributes passed in Python. This feature is available only with Python 3.5 and beyond[1].

Docstrings should be added to the class definition in a similar fashion to how they were added to the function definitions from previous lessons. The docstrings for different calls in Python have different elements, so do not contain the same information. For instance, the Class CD will have properties and methods while the method load\_inventory will have a description, arguments and a return.

Everything in Python is an object, and thus all have some inheritance from a base class. The assignments then have dealt with this same material implicitly throughout the course. However, now the mechanism of inheritance is exposed.

Script Creation

The baseline has been provided as a starter in the module. There are “TODO” markers and pseudocode through that represent the code and functionality that needed to be added. Some of the more straightforward aspects of writing the code will be identified and handled first. Replacing the header information and collecting the field names to be copied outside the docstring is the first step.

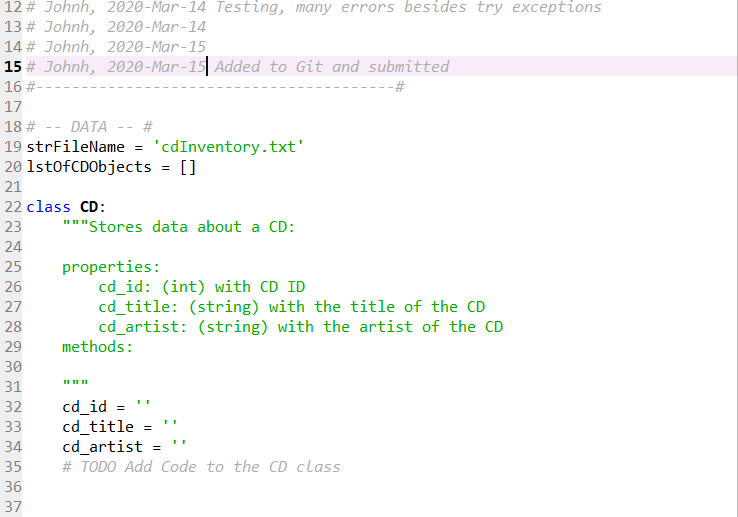


Figure 1 Header replaced from the template. Properties copied from docstrings.

Data flow needed to be established. In previous starter files the menu exposition to the user was provided. Here is was necessary to rewrite that functionality into the IO class. Class docstrings were updated as the classes were updated. In order to keep the code added to the starter package consistent, the code was copied from the starter for assignment06 whenever possible[12].

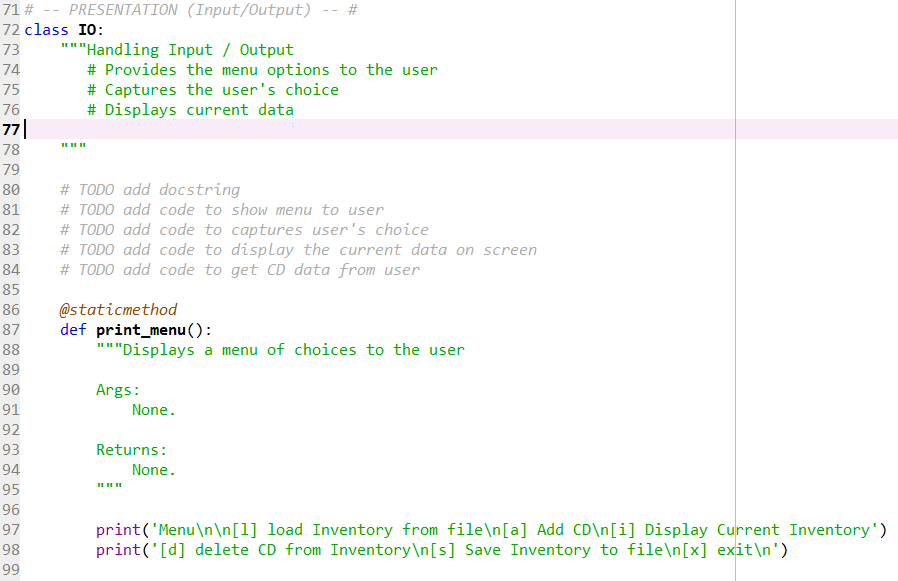


Figure 2 The print menu and other function definitions were added to the class definitions here in the same manner as was done in Assignment 6.

Much of the basic functionality was borrowed from the previous assignments. Again, the specifics of the syntax are only as important as the organization of the code. The class, functions, docstrings, methods, and OOP in general is more about enabling the organization than employing the syntax of each programming convention.

The question of test script writing came up in the last “lecture” but was not addressed. Considering the complexity, at least relative complexity, the testing is not a simple enough task to accomplish without some structure. Notes were taken on the testing, and some are presented here for reference, and use of the developer.

Test1: Script runs in Spyder, menu is displayed, file path is set to C:\Users\johnh\FundaMental\Assignments\Assignmnet08, no ‘cdInventory.txt’ file exists, ‘i’ is selected, the output text formatting is returned, no CD entries are contained in the output, prompt returns to menu, ‘x’ is selected, application exits without error.

Test2: Script runs in Spyder, menu is displayed, file path unchanged, no ‘cdInventory.txt’ file exists, ‘d’ is selected, empty inventory is displayed, “Which ID..” prompt appears, ‘34’ is entered, “Could not find this CD!” is displayed, ‘a’ is attempted to be entered, but the enter key does not execute when ‘a’ is typed. This error type is unknown, script can not be killed, Spyder is closed.

Test3: Script runs in Spyder, menu is displayed, file path unchanged, no ‘cdInventory.txt’ file exists, ‘a’ is selected, errors thrown “TypeError: ask() missing 1 required positional argument: 'self'” line 325, which is “userCD = IO.ask()”, removed ‘self’ from the def is now “def ask():”. Logical error discovered, passing the user entered data is not assigned correctly in CD class at the object instantiation level.

A partial rewrite of the script was necessary. The methods, show\_inventory, delete\_cd, save\_inventory, # read\_file, I may not keep this method, all needed to be refactored to handle the object being passed. The rewrite of the show\_inventory is shown in the figure.

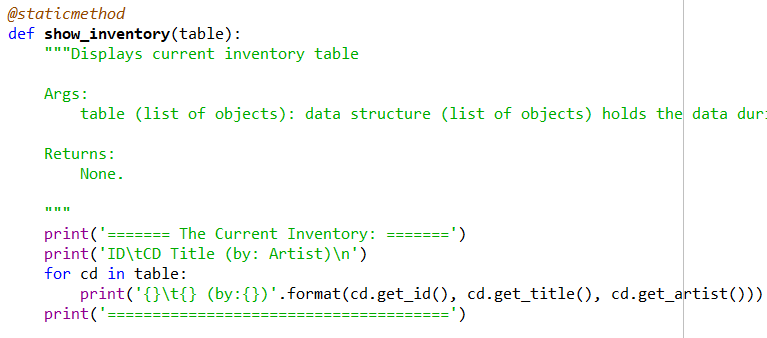


Figure 3 Simple changes to loops should be expected given the reliance on data type for the iteration mechanism.

The rewrites were all of a similar flavor, iterating over the objects, using the getter methods. There were surprisingly only two total calls to the CD class needed because of the looping. Testing was resumed shortly after the rewrites were finished. After simple syntax and logical errors were sorted, the previous test scripts were run without error. Additional error handling was added for user input variable typing, and other IO operations. The intent was to allow a user to access the script, whether the .txt exists or not, and whether the .txt was empty, or prepopulated. In any case, for the user, the script loads and displays the inventory. Some other “errors” that were “handled” in previous iterations were removed. An actual CD inventory could contain two copies of the same album.

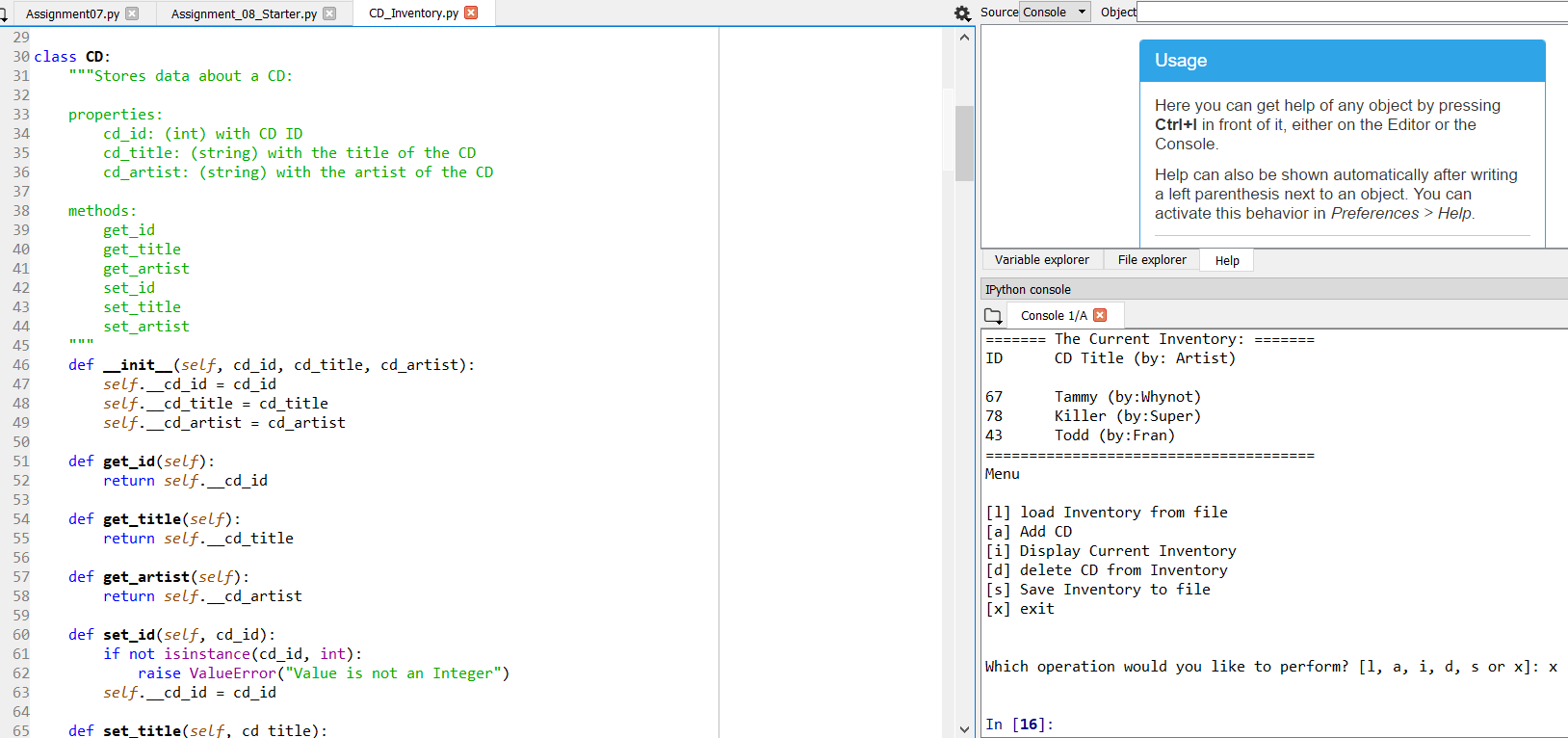


Figure 4 Script executes as expected in Spyder.

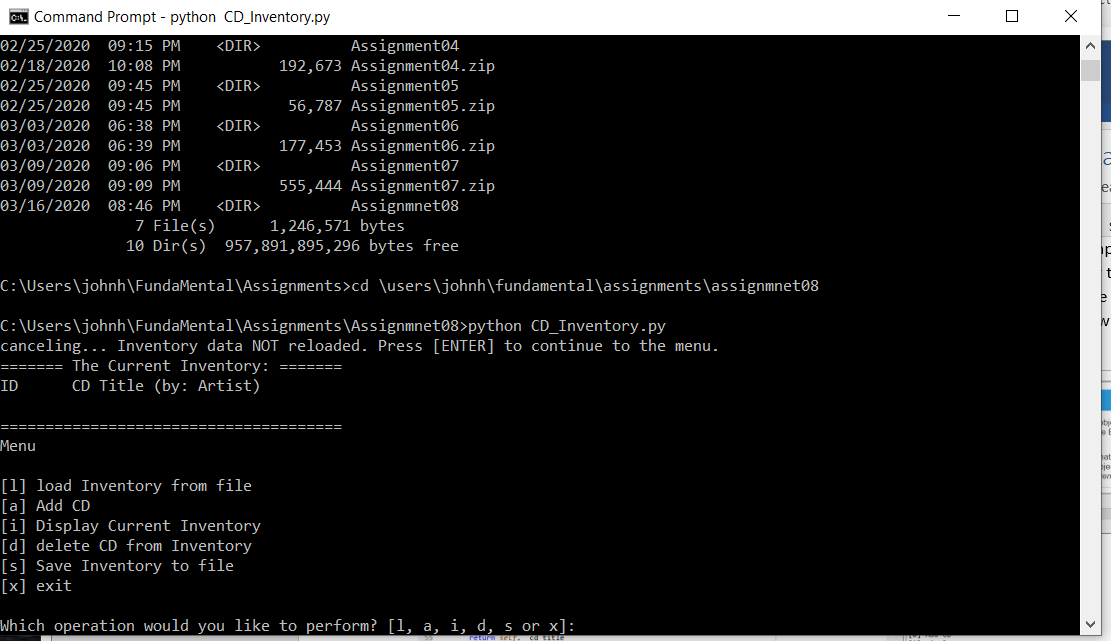


Figure 5 Testing should include not just the functionality of the application but the environment as well. Here the file path has a misspelling.

Summary

Robust applications employee many layers of abstraction. The abstraction layers can take many forms, from the representation needed for the UI so that a user can input data properly, to a data model representation, to the database specific implementation of the data model. Object Oriented Programming is capable of creating a fundamental abstraction and structure that can enable other hierarchical abstraction layer creation in service of creating a more robust application or system. Classes are used to define the structure of the object that will be instantiated in the body of the application. Classes use methods, properties, attributes and fields as elements to define the structure of the objects. Constructor methods getters and setters, can be used to create a more “pythonic” way of passing values to instantiate the object. The Python base Class Object is the class which defines the basic structure that all Classes inherit. Several conventions, like the ‘self’ almost keyword, and the dunder \_\_init\_\_ create a framework for passing values and manipulating objects.

Appendix

Planet B Syntax Highlighter

1. #------------------------------------------#
2. # Title: CD\_Inventory.py
3. # Desc: Create a script that asks the user to select their choice read,
4. #       write, save, or delete CD inventory data, storing them in a text file.
5. #       The functionality should be accomplished by the use of classes.
6. #       Exceptions should be handled, docstrings updated, read/write functions.
7. #
8. # Change Log:
9. # Johnh, 2020-Mar-13 Script Created from starter assignment08 .py file
10. # Johnh, 2020-Mar-13 Assignment and background material consumed in preparation
11. #                    for development
12. # Johnh, 2020-Mar-14 Added properties cd\_id, cd\_title, and CD\_artist to CD
13. #                    class defintion
14. # Johnh, 2020-Mar-14 Analyzed the organization of the code, imported reusable
15. #                    class and function sections from previous assignments
16. # Johnh, 2020-Mar-14 Analyzed main body, reused while format, no variables
17. #                    handled in the loop, all data handled in classes.
18. # Johnh, 2020-Mar-14 Initial testing, script runs in Spyder.
19. # Johnh, 2020-Mar-15 Errors found in 'a', 'l', 's' functions.
20. # Johnh, 2020-Mar-15 Display method modified to handle the object instantiations
21. # Johnh, 2020-Mar-16 Testing, error handling, docstrings, readability
22. # Johnh, 2020-Mar-16 Added to Git and submitted
23. #----------------------------------------#
24. **from** os **import** path
26. # -- DATA -- #
27. strFileName = 'cdInventory.txt'
28. lstOfCDObjects = []
30. **class** CD:
31. """Stores data about a CD:
33. properties:
34. cd\_id: (int) with CD ID
35. cd\_title: (string) with the title of the CD
36. cd\_artist: (string) with the artist of the CD
38. methods:
39. get\_id
40. get\_title
41. get\_artist
42. set\_id
43. set\_title
44. set\_artist
45. """
46. **def** \_\_init\_\_(self, cd\_id, cd\_title, cd\_artist):
47. self.\_\_cd\_id = cd\_id
48. self.\_\_cd\_title = cd\_title
49. self.\_\_cd\_artist = cd\_artist
51. **def** get\_id(self):
52. **return** self.\_\_cd\_id
54. **def** get\_title(self):
55. **return** self.\_\_cd\_title
57. **def** get\_artist(self):
58. **return** self.\_\_cd\_artist
60. **def** set\_id(self, cd\_id):
61. **if** **not** isinstance(cd\_id, int):
62. **raise** ValueError("Value is not an Integer")
63. self.\_\_cd\_id = cd\_id
65. **def** set\_title(self, cd\_title):
66. **if** **not** isinstance(cd\_title, str):
67. **raise** ValueError("Value is not a String")
68. self.\_\_cd\_title = cd\_title
70. **def** set\_artist(self, cd\_artist):
71. **if** **not** isinstance(cd\_artist, str):
72. **raise** ValueError("Value is not a String")
73. self.\_\_cd\_artist = cd\_artist

76. # -- PROCESSING -- #
77. **class** FileIO:
78. """Processes data to and from file:
80. properties:
82. methods:
83. save\_inventory(file\_name, lst\_Inventory): -> None
84. load\_inventory(file\_name): -> (a list of CD objects)
85. load\_table(lines, table)
87. """
89. @staticmethod
90. **def** load\_inventory(file\_name, table, STARTUP = False):
91. """Manage data from file to a list of dictionaries in runtime
93. Reads the data from file file\_name into a list of objects, table
95. Args:
96. file\_name (string): name of file used to read the data from
97. table (list of objects): holds the data during runtime
99. Returns:
100. table
101. """
102. **if** **not** STARTUP:
103. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
104. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled')
105. **else**:
106. strYesNo = "yes"
108. **if** strYesNo.lower() == "yes":
109. **if** path.exists(file\_name):
110. **print**('reloading...')
111. file = open(file\_name, "r")
112. lines = []
113. **for** line **in** file:
114. line = line.strip()
115. lines.append(line)
116. file.close()
117. **if** len(lines) > 0:
118. FileIO.load\_table(lines, table)
119. IO.show\_inventory(table)
120. **else**:
121. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
122. IO.show\_inventory(table)
124. **else**:
125. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
126. IO.show\_inventory(table)
127. **else**:
128. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
129. IO.show\_inventory(table)
130. **return** table

133. @staticmethod
134. **def** save\_inventory(file\_name, inventory):
135. """Writes the inventory of IDs, CD Names, and Artists to a text file
137. Args:
138. file\_name (string): The name of the file that it will write to
139. table (list of dict): 2D data structure (list of dicts) holds the data during runtime
141. Returns:
142. None but saves a file in the directory of the python script
144. """
146. IO.show\_inventory(inventory)
147. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
149. **if** strYesNo == 'y':
150. # save data
151. file = open(file\_name, 'w')
152. **for** cd **in** inventory:
153. cd\_properties = [str(cd.get\_id()), cd.get\_title(), cd.get\_artist()]
154. file.write(','.join(cd\_properties) + '\n')
155. file.close()
156. **else**:
157. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
159. @staticmethod
160. **def** load\_table(lines, table):
161. table.clear()
162. **for** line **in** lines:
163. cd = line.split(",")
164. cd\_id = int(cd[0])
165. cd\_title = cd[1]
166. cd\_artist = cd[2]
167. table.append(CD(cd\_id,cd\_title,cd\_artist))

170. # -- PRESENTATION (Input/Output) -- #
171. **class** IO:
172. """Class definition to handle Input / Output
173. Handling Input / Output
175. Properties:
176. None.
178. Methods:
179. print\_menu(): -> Provides the menu options to the user
180. menu\_choice(): -> Captures the user's choice
181. show\_inventory(table): -> Displays current data
182. """
184. @staticmethod
185. **def** print\_menu():
186. """Displays a menu of choices to the user
188. Args:
189. None.
191. Returns:
192. None.
193. """
195. **print**("Menu\n\n" +
196. "[l] load Inventory from file\n" +
197. "[a] Add CD\n" +
198. "[i] Display Current Inventory")
200. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
202. @staticmethod
203. **def** menu\_choice():
204. """Gets user input for menu selection
206. Args:
207. None.
209. Returns:
210. choice (string): a lower case string of the users input l, a, i, d, s or x
212. """
213. choice = ' '
214. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
215. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
216. **print**()  # Add extra space for layout
217. **return** choice
219. @staticmethod
220. **def** show\_inventory(table):
221. """Displays current inventory table
223. Args:
224. table (list of objects): data structure (list of objects) holds the data during runtime.
226. Returns:
227. None.
229. """
230. **print**('======= The Current Inventory: =======')
231. **print**('ID\tCD Title (by: Artist)\n')
232. **for** cd **in** table:
233. **print**('{}\t{} (by:{})'.format(cd.get\_id(), cd.get\_title(), cd.get\_artist()))
234. **print**('======================================')
236. @staticmethod
237. **def** delete\_cd(table):
238. """Deletes a CD row from the table
240. Args:
241. intIDDel (int): ID indicates user entry to delete
242. table (list of objects): data structure (list of objects) that holds the data during runtime
244. Returns:
245. table (list of objects): data structure (list of objects) that holds the data during runtime
246. """
247. IO.show\_inventory(table)
248. # ask user which ID to remove
249. intIDDel = int(input('Which ID would you like to delete? ').strip())
250. # search thru table and delete CD
251. blnCDRemoved = False
252. **for** cd **in** table:
253. **if** cd.get\_id() == intIDDel:
254. table.remove(cd)
255. blnCDRemoved = True
256. **break**
257. **if** blnCDRemoved:
258. **print**('The CD was removed')
259. **else**:
260. **print**('Could not find this CD!')
261. IO.show\_inventory(table)
262. **return** table
264. @staticmethod
265. **def** ask():# Code to get CD data from user
266. """Ask user for new ID, CD Title and Artist, values are
267. get and set in CD class
269. Args:
270. None
272. Returns:
273. dicRow (CD object):  An object with cd\_id(int), cd\_title(str),
274. cd\_artist(str) fields to hold the id, title, and artist name respectively.
275. """
277. flag = True
278. **while** flag:
279. **try**:
280. objcd\_id = int(input('Enter ID: ').strip())
281. flag = False
282. **except** ValueError:
283. **print**("INVALID INPUT enter an Integer")
285. objcd\_title = input('What is the CD\'s title? ').strip()
286. objcd\_artist = input('What is the Artist\'s name? ').strip()
287. dicRow = CD(objcd\_id,objcd\_title,objcd\_artist)
289. **return** dicRow
291. # -- Main Body of Script -- #
292. FileIO.load\_inventory(strFileName,lstOfCDObjects, True)
293. IO.print\_menu()
294. strChoice = IO.menu\_choice()
295. **while** strChoice != 'x':
297. **if** strChoice == 'i':                    # show user current inventory
298. IO.show\_inventory(lstOfCDObjects)
300. **elif** strChoice == 'a':                   # let user add data to the inventory
301. userCD = IO.ask()
302. lstOfCDObjects.append(userCD)
303. IO.show\_inventory(lstOfCDObjects)
305. **elif** strChoice == 'd':                  # Let user delete data from the Inventory
306. lstOfCDObjects = IO.delete\_cd(lstOfCDObjects)
308. **elif** strChoice == 's':                  # let user save inventory to file
309. FileIO.save\_inventory(strFileName, lstOfCDObjects)
311. **elif** strChoice == 'l':                  # let user load inventory from file
312. lstOfCDObjects = FileIO.load\_inventory(strFileName, lstOfCDObjects)
314. **else**:
315. **print**('General Error')#
317. IO.print\_menu()
318. strChoice = IO.menu\_choice()

References

1. FDN\_Py\_Module\_08
2. Assignment\_08
3. https://www.geeksforgeeks.org/class-instance-attributes-python/
4. Michael Dawson Python Programming
5. https://youtu.be/lkXfgP-fAkY
6. https://youtu.be/KZdFvyCOLUQ
7. https://youtu.be/qvRyls8NX-E
8. https://youtu.be/QuwU34OT4XA
9. https://youtu.be/swAXTwW6xoA
10. https://youtu.be/5MM6IaESdQ0
11. https://realpython.com/python3-object-oriented-programming/
12. Assignment06\_Starter.py
13. https://www.programiz.com/python-programming/property