Christopher J. Scott

9-Mar-2021

IT FDN 110: Introduction to Programming (Python)

Assignment 08

Introduction to Object Oriented Programming in Python: Working with Classes, Constructors, Attributes, Properties and Decorators

# Introduction

Module 08 brought about a different way of thinking about programming and introduced the concepts around Object Oriented Programming (OOP). According to module 08, “OOP has a reputation of being difficult to understand and overly complicated”, and I found that to be true for the most part. While classes were already previously introduced, the introduction of constructors, attributes, properties, methods and decorators proved to be a “tough nut to crack” for me personally. By the end of module 08, however, a user is able to incorporate these lessons into a program that expands on the CD Inventory project that we have been developing throughout the entirety of the class.

# Basics of Classes

Classes were first introduced as a way to create a blueprint for a particular object. This blueprint can be used multiple times to generate a new instance, and that particular instance will be the result of executing the blueprint with any particular attributes.

The classes used in this Assignment are class CD, class FileIO and class IO which correspond to the format used to separate concerns: data, processing and presentation (input/out) respectively.



Figure . 3 classes defined in Assignment 08

Within each class, fields, constructors, properties and methods can all be defined to govern what is done to the object with input attributes.

# Constructors

Constructors are a specialized method within Python that run during the initial creation of an object. In order to utilize constructors for this assignment, the \_\_init\_\_() method is called which serves to pass any included arguments through.

Lines 28 through 37 in Figure 2 below show the \_\_init\_\_() method being used to instantiate the parameters of the each CD within the inventory (ID, title and artist).

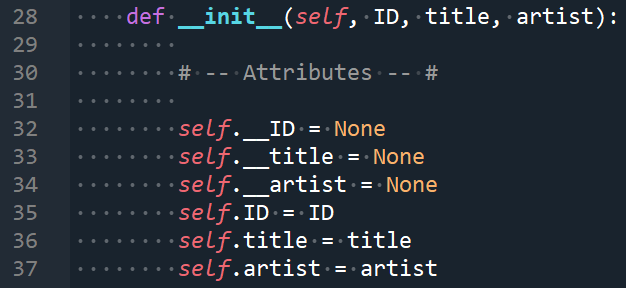


Figure . \_\_init\_\_() method being used to instantiate CD objects

NOTE: the keyword self that is used here is not an *official* keyword within python but it is widely accepted as convention within the community and is universally accepted as the first reference to an object.

# Attributes

Attributes are internal fields or variables within a class that hold data and are not controlled during runtime from outside the class unless you specifically provide code to validate values before they are assigned. Those special methods are called Properties.

# Properties

The special methods used to control validity of values are called “getters” and “setters” and are used to make attributes private and enforce interaction with built in control mechanisms.

The getter methods allow a user to add code to format fields or attributes data while setter methods allow a user to add validation and error handling statements. A simple example of this is seen in Figure 3 below where the error handling for the property ID is provided within the @ID.setter decorator.

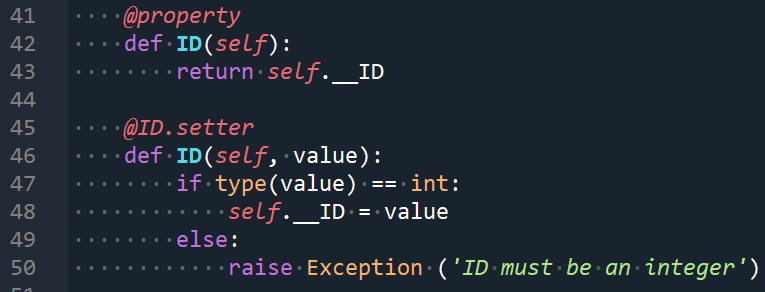


Figure . Getter and Setter used for ID of CD

# Methods

Methods can be thought of as functions contained within a class. A key difference between a method and a typical function is that a method submits a reference to its own object so the first attribute that is supplied to a method is the “self” reference. This is important because a user will need to ensure that the other attributes are supplied in the correct order.

# Decorators

Decorators are simply “wrappers” that go around a function or method and change the behavior or scope of those functions/methods. Depending on the decorator, certain functionality is assigned to the enclosed functions and custom decorators can be created if desired.

# Putting it all together in Assignment 08

In order to apply the knowledge learned in Module 08, we first started with the provided starter code that contained pseudo code and specific instructions. Luckily, much of the methods were already created in previous assignments, they just needed to be modified to call the correct classes with the correct attributes.

It can be seen that error handling can be added at various points within the program including within properties, methods or the main body.

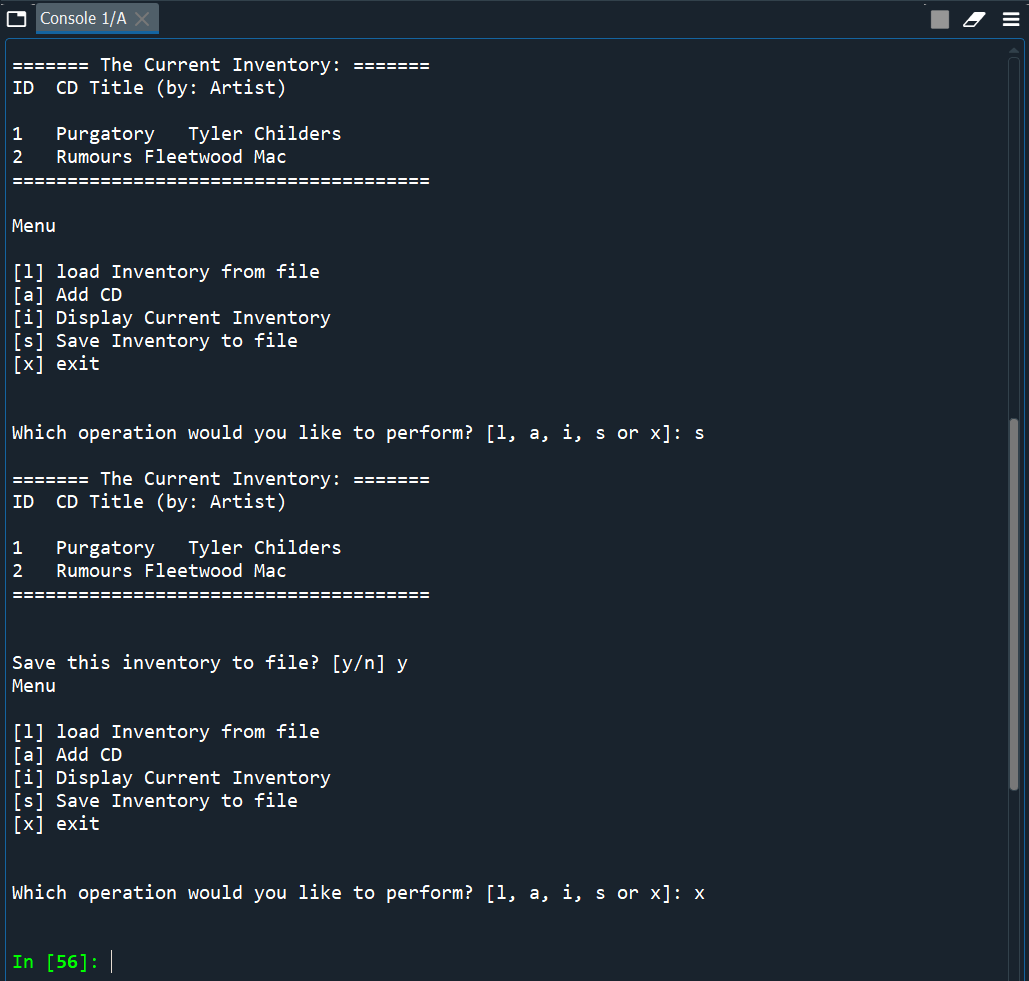
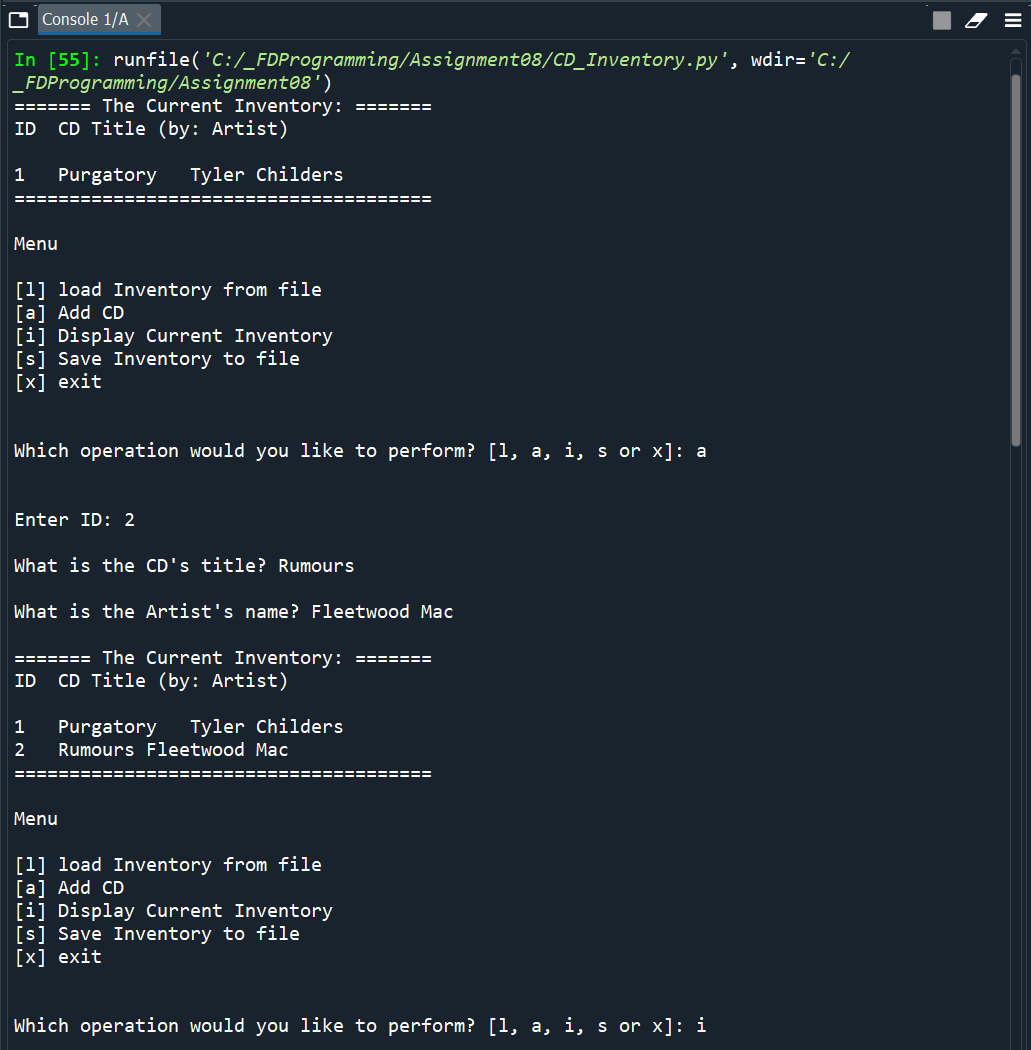


Figure . Result of running and interacting with CDInventory.py within Spyder

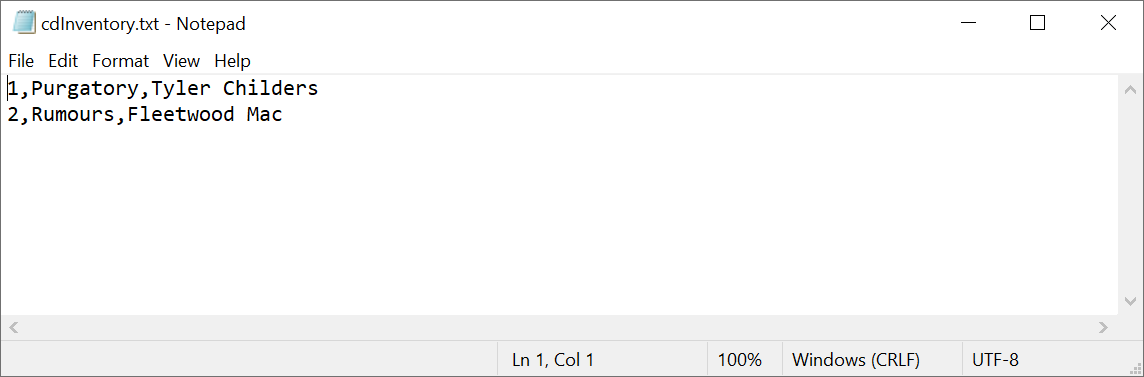


Figure . Output .txt file after running save command

# Summary

By completing Module 8 and associated Lab exercises, a user is able to create a program that utilizes classes, getters and setters, and methods to allow a user to load, save, add, display and exit a CD inventory. As was the case in previous modules, a repository that contains this specific CDInventory.py for Assignment 08 can be found at <https://github.com/cscott08vt/Assignment08>.

# Appendix

1. #------------------------------------------#
2. # Title: Assignmen08.py
3. # Desc: Assignnment 08 - Working with classes
4. # Change Log: (Who, When, What)
5. # DBiesinger, 2030-Jan-01, created file
6. # DBiesinger, 2030-Jan-01, added pseudocode to complete assignment 08
7. # CScott, 2021-Mar-07, Added IO class, replaced pseudocode with actual code
8. #------------------------------------------#
9. # -- DATA -- #
10. strFileName = 'cdInventory.txt'
11. lstOfCDObjects = []
13. **class** CD:
14. """Stores data about a CD:
16. properties:
17. ID: (int) with CD ID
18. title: (string) with the title of the CD
19. artist: (string) with the artist of the CD
20. methods:
21. \_\_str\_\_(): -> formatted string of CD objects
22. file\_str(): -> formatted string of CD objects
23. """
24. # -- Fields -- #
25. # -- Constructor -- #
27. **def** \_\_init\_\_(self, ID, title, artist):
29. # -- Attributes -- #
31. self.\_\_ID = None
32. self.\_\_title = None
33. self.\_\_artist = None
34. self.ID = ID
35. self.title = title
36. self.artist = artist
38. # -- Properties -- #
40. @property
41. **def** ID(self):
42. **return** self.\_\_ID
44. @ID.setter
45. **def** ID(self, value):
46. **if** type(value) == int:
47. self.\_\_ID = value
48. **else**:
49. **raise** Exception ('ID must be an integer')
51. @property
52. **def** title(self):
53. **return** self.\_\_title
55. @title.setter
56. **def** title(self, value):
57. self.\_\_title = value
59. @property
60. **def** artist(self):
61. **return** self.\_\_artist
63. @artist.setter
64. **def** artist(self, value):
65. self.\_\_artist = value
67. # -- Methods -- #
69. **def** \_\_str\_\_(self):
70. **return** '{}\t{}\t{}'.format(self.ID, self.title, self.artist)
72. **def** file\_str(self):
73. **return** '{},{},{}\n'.format(self.ID, self.title, self.artist)
75. # -- PROCESSING -- #
76. **class** FileIO:
77. """Processes data to and from file:
79. properties:
81. methods:
82. save\_inventory(file\_name, table): -> None
83. load\_inventory(file\_name): -> (a list of CD objects)
85. """
86. # -- Fields -- #
87. # -- Constructor -- #
88. # -- Properties -- #
89. # -- Methods -- #
90. @staticmethod
91. **def** save\_inventory(file\_name, table):
92. """Function to manage exporting data from list of lists to a file
94. Writes the data to file identified by file\_name from a 2D table
95. (list of dicts) table one line in the file represents one dictionary row in table.
97. Args:
98. file\_name (string): name of file that the data is written to
99. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
101. Returns:
102. None.
103. """
104. with open(file\_name, 'w') as objFile:
105. **for** line **in** table:
106. objFile.write(line.file\_str())
108. @staticmethod
109. **def** load\_inventory(file\_name, table):
110. """Function to manage data ingestion from serialized file to a list of dictionaries
112. Reads the data from file identified by file\_name into a 2D table
113. (list of lists) table one line in the file represents one dictionary row in table.
115. Args:
116. file\_name (string): name of file used to read the data from
117. table (list of lists): 2D data structure (list of dicts) that holds the data during runtime
119. Returns:
120. None.
121. """
122. **try**:
123. table.clear()  # this clears existing data and allows to load data from file
124. with open(file\_name, 'r') as objFile:
125. **for** line **in** objFile:
126. data = line.strip().split(',')
127. cd = CD(int(data[0]),data[1],data[2])
128. table.append(cd)
129. **except** FileNotFoundError as e:
130. with open(file\_name, 'w') as objFile:
131. **print**('ERROR:'+ e.\_\_doc\_\_ +' An empty file called ' + file\_name + ' was created within the working directory')
132. **except** EOFError as e:
133. **print**('The data file is empty! Current inventory is blank')
135. # -- PRESENTATION (Input/Output) -- #
136. **class** IO:
137. """Inputs and outputs data to/from user:
139. properties:
141. methods:
142. print\_menu(): -> None
143. menu\_choice(): -> (string) User choice
144. show\_inventory(table): -> None
145. ID\_title\_artist\_add(): -> intID, strTitle, strArtist
147. """
148. # -- Fields -- #
149. # -- Constructor -- #
150. # -- Properties -- #
151. # -- Methods -- #
152. @staticmethod
153. **def** print\_menu():
154. """Displays a menu of choices to the user
156. Args:
157. None.
159. Returns:
160. None.
161. """
163. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
164. **print**('[s] Save Inventory to file\n[x] exit\n')
166. @staticmethod
167. **def** menu\_choice():
168. """Gets user input for menu selection
170. Args:
171. None.
173. Returns:
174. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
176. """
177. choice = ' '
178. **while** choice **not** **in** ['l', 'a', 'i', 's', 'x']:
179. choice = input('Which operation would you like to perform? [l, a, i, s or x]: ').lower().strip()
180. **print**()  # Add extra space for layout
181. **return** choice
183. @staticmethod
184. **def** show\_inventory(table):
185. """
186. Displays current inventory table
188. Args:
189. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
191. Returns:
192. None.
194. """
195. **print**('======= The Current Inventory: =======')
196. **print**('ID\tCD Title (by: Artist)\n')
197. **for** row **in** table:
198. **print**(row)
199. **print**('======================================')
200. **print**() # Add extra space for layout
202. @staticmethod
203. **def** ID\_title\_artist\_add():
204. """Gets user input for specific CD to add
206. Args:
207. None.
209. Returns:
210. CD\_info (list): list containing CD ID, title and artist
212. """
213. **while** True:
214. **try**:
215. intID = int(input('Enter ID: ').strip())
216. **break**
217. **except** ValueError as e:
218. **print**('ERROR: ID that was entered is not of type: integer')
219. **print**(e.\_\_doc\_\_)
221. strTitle = input('What is the CD\'s title? ').strip()
222. strArtist = input('What is the Artist\'s name? ').strip()
223. **print**() # Add extra space for layout
224. **return** intID, strTitle, strArtist
226. # 1. When program starts, read in the currently saved Inventory
227. FileIO.load\_inventory(strFileName, lstOfCDObjects)
228. IO.show\_inventory(lstOfCDObjects)
230. # 2. Start main loop
231. **while** True:
232. # 2.1 Display Menu to user and get choice
233. IO.print\_menu()
234. strChoice = IO.menu\_choice()
235. # 3. Process menu selection
236. # 3.1 process exit first
237. **if** strChoice == 'x':
238. **break**
239. # 3.2 process load inventory
240. **if** strChoice == 'l':
241. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
242. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
243. **if** strYesNo.lower() == 'yes':
244. **print**('reloading...')
245. FileIO.load\_inventory(strFileName, lstOfCDObjects)
246. IO.show\_inventory(lstOfCDObjects)
247. **else**:
248. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
249. IO.show\_inventory(lstOfCDObjects)
250. **continue**  # start loop back at top.
251. # 3.3 process add a CD
252. **elif** strChoice == 'a':
253. # 3.3.1 Ask user for new ID, CD Title and Artist
254. intID, strTitle, strArtist = IO.ID\_title\_artist\_add()
255. CDObj = CD(intID, strTitle, strArtist)
256. # 3.3.2 Add item to the table
257. lstOfCDObjects.append(CDObj)
258. # 3.3.3 Display current inventory after adding CD
259. IO.show\_inventory(lstOfCDObjects)
260. **continue**  # start loop back at top.
261. # 3.4 process display current inventory
262. **elif** strChoice == 'i':
263. IO.show\_inventory(lstOfCDObjects)
264. **continue**  # start loop back at top
265. # 3.5 process save inventory to file
266. **elif** strChoice == 's':
267. # 3.6.1 Display current inventory and ask user for confirmation to save
268. IO.show\_inventory(lstOfCDObjects)
269. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
270. # 3.6.2 Process choice
271. **if** strYesNo == 'y':
272. # 3.6.2.1 save data
273. FileIO.save\_inventory(strFileName, lstOfCDObjects)
274. **else**:
275. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
276. **continue**  # start loop back at top.
277. # 3.6 catch-all should not be possible, as user choice gets vetted in IO, but to be safe:
278. **else**:
279. **print**('General Error')

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