Meredith Bailey

March 13, 2020

Foundations of Programming: Python

Assignment 08

Assignment 8 Knowledge Document

# Introduction

This week, we learned about classes and their main components: fields, constructors, attributes, properties, and methods. There are other things that can be used in classes to further define how a piece of code will function, such as double underscores, decorators, and type hints.

# Classes: Attributes, Constructors, Properties, and Methods

Classes package the data and functionality of an object, and objects are an instance of a class. Each object uses the same class as a blueprints, but generates its own copy in memory. Classes have five main components: constructors, fields, attributes, properties, and methods.

Constructors are a method that is invoked when creating an object. They are a convenient way to ensure proper data types in the fields. They also allow for a pre-population, or default value setting, to the fields, similar to using default values in functions. The Python constructor method is \_\_init\_\_(). This method is called each time the object is created.

Fields are the data stores of a class, and get created the same way as variables. Attributes are internal data stores of a class, and can be created implicitly by the constructor function. However, the implicit creation of attributes is not usually seen in other languages. Attributes are just like variables – there is no limitation on the data they can store. In order to control what kind of data is stored in attributes, you have to write specific code to validate the attributes’ values before they are assigned. This is done by making the attributes private (which is done by adding the double underscore to the beginning of the variable name) and by using Properties.

Properties are methods that have control mechanisms built in to control the validity of values assigned to attributes. Two are created for each attribute – one to access the value, or a “getter”, and one to set the value, or a “setter”. Both properties are titled with the attribute name itself. The getters take no arguments other than ‘self’, and the setters take in ‘self’ as well as the new attribute value. In order to identify which property performs which task, decorators are used. The @property decorator is used for getters, and the @attribute\_name.setter decorator is used for setters. When defining properties in the class, the getter must be defined first, then the setter. In object-oriented programming, it is considered best practice to only work with data in a class through a method or property, as this creates a layer of abstraction. Abstraction means that other programmers do not have to worry about the inner-workings of the class when they use it. In Python, everything is open and accessible by design, so “private” attributes are technically accessible – they are more considered private by convention.

Methods are like functions can be invoked by calling the method’s name. Most languages include a method that returns some or all of the object’s data as a string. The built-in method in Python is called \_\_str\_\_. In Python, if the programmer does not define their own \_\_str\_\_, the class will simply inherit the method from the base class object. Statis methods are run on the class level and not the instance level. These static methods, along with class fields, can be used to keep track of information that concerns the class. The @staticmethod decorator is used to define static methods. When a class focuses on processing data, static methods are usually used. If the class focuses on storing data, instance methods can be used. As with attributes, private methods can be created, again indicated by two leading underscores.

Lastly, when defining attributes, type hints show what type of data is expected in attributes and what type is returned. In many languages, type definitions are necessary and enforced. In Python, these are not enforced and have a more “informal” character – they simply help to inform other developers what is expected.

# Using Classes: Attributes, Properties, and Methods in the CD Inventory

When creating this program, I started from a “skeleton” file that had only pseudocode, then added code elements to each section of pseudocode. I copied and pasted some functions from our previous assignments, ensuring to change variable names and modify function arguments where necessary. I first created the CD class, wrote the getter and setter properties, and created the constructor and \_\_str\_\_ functions. I then copied and pasted the functions from the previous assignment. The last thing I did was write the code in the main body, which I actually did from scratch and did not copy and paste. I did this because I assumed that the main body of the code would change significantly now that I had my CD class, and I thought that it would be easier to create it from scratch than to try to edit in all the references to my new CD class. I was worried that I would get a bunch of errors due to the new way of storing and referencing data. But actually, I didn’t! It was a huge relief that all my pieces of code seemed to work together well.

I wanted to challenge myself and add a new feature to the code: assigning ID numbers automatically. As a user of this program, I would not really care what the ID numbers of each CD were, I would only care that each CD was stored properly. I think it would be more of a hassle to come up with unique ID numbers myself, and try to remember where I previously left off in the ID numbering sequence each time I came back to the program. Therefore, I originally wanted to use a counter (numCDs) in conjunction with a static method that would run upon class instance construction to automatically assign the CD’s ID number.

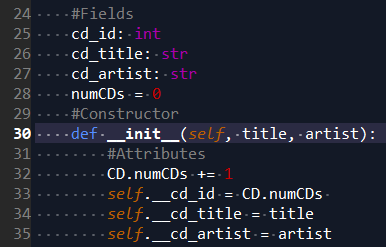


Figure : Trying to use a counter variable with a static method.

However, if I exited out of the program and then reran the program to add more CDs to the inventory, the counter would start over at 1. This can be seen below. After evaluating the logic of my code, this made sense, since my new CD was the first time an instance of the CD class was created in this program’s runtime. Restarting the number sequence every time the program ran was undesirable, because it created duplicate ID numbers and prevented the CD inventory from become one, cohesive inventory.

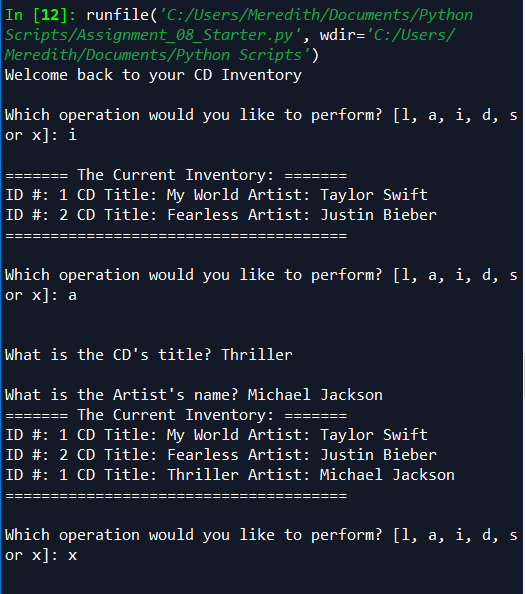


Figure : Counter variable would restart at 1 upon program restart.

Therefore, I did not use a counter, but rather passed in the list of CDs to the class, and used its length to determine what the next ID number should be.

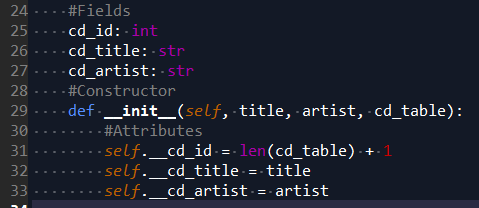


Figure : Replacing the counter variable with the length of the cd\_table.

This fixed the problem of not having sequential ID numbers after exiting and rerunning the program, which can be seen below.

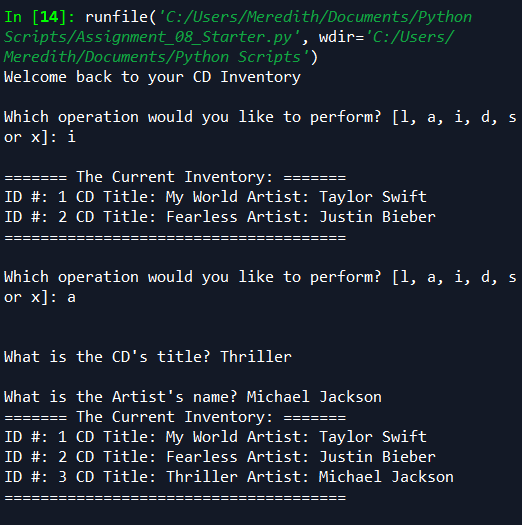


Figure : This appeared to solve the problem that was occurring upon program restart...

However, this created a new problem: after CDs were deleted from the table, the new ID numbers were wrong. Again, based on the code’s logic, this made sense. ID numbers were being assigned based on the length of the list of CDs, and an object had just been removed from the list, decreasing the length of the list. This meant that the sequence of numbers would be incorrect.

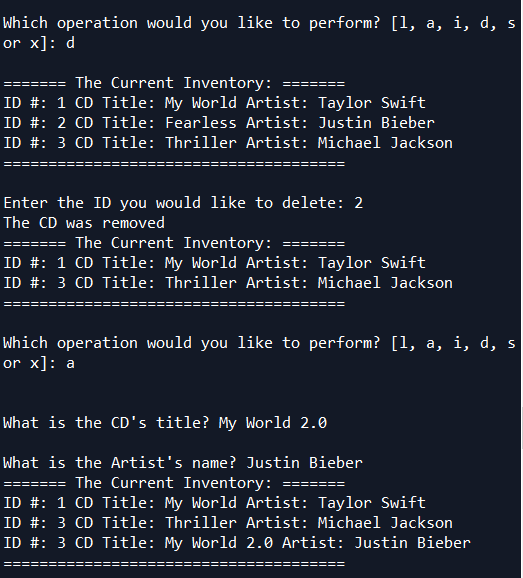


Figure : A new problem was identified in ID sequencing after deletion.

Basing the ID number on the length of the table solved my problem of not having sequential numbers upon restarting the program. But now, I wanted a way to re-sequence the ID numbers after a CD had been deleted. To do this, I added the following for loop to the end of my delete\_cd() function:



Figure : Added a for loop to the delete function to reassign numbers after deletion.

This then re-assigned ID numbers to each CD based on that CD’s position in the list. This yielded the following result:

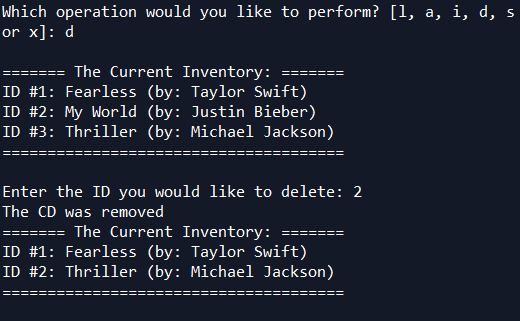


Figure : ID numbers remain sequential, even after a CD in the middle of the list is deleted.

The CDs’ ID numbers were successfully re-sequenced after deletion! So, the final result of how I decided to handle ID numbers is as follows: first, the program automatically assigns the ID number that should come next based on the length of the CD table. If this sequence of ID numbers is disrupted, such as when a user deletes a CD from the middle of the list, the delete function will automatically reassign ID numbers based on the position of each CD in the list. I realize that with a very long list, this could take a long time. Perhaps it would be better to only reassign IDs for CDs that were listed after the CD that was deleted – this would reduce redundant processing on CDs whose ID numbers were unaffected by the deletion.

# Summary

This week, we learned the main components of classes and learned how to use them. We practiced the real-life scenario of migrating pieces of code from one program to another, and had to edit the way the main body of the code interacted with a new method of storing data.

# Appendix

Source code from this assignment (created using [PlanetB’s](https://planetb.ca/syntax-highlight-word) website):

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. # MBailey, 2020-Mar-13, added CD and DataProcessing classes, added code to main body, added code to FileIO and IO classes. Program runs successfully.
8. # MBailey, 2020-Mar-15 added docstrings and comments where necessary.
9. #------------------------------------------#
11. # -- DATA -- #
12. strFileName = 'cdInventory.txt'
13. lstOfCDObjects = []
14. **import** pickle
16. **class** CD:
17. """Stores data about a CD:
19. properties:
20. cd\_id: (int) getter and setter
21. cd\_title: (string) getter and setter
22. cd\_artist: (string) getter and setter
23. methods:
24. \_\_str\_\_: returns formatted string of data stored in attributes
25. """
26. #Fields
27. cd\_id: int
28. cd\_title: str
29. cd\_artist: str
30. #Constructor
31. **def** \_\_init\_\_(self, title, artist, cd\_table):
32. """constructor for CD class
33. args: title (str)
34. artist (str)
35. cd\_table (list), used for assigning sequential ID numbers
36. returns: none"""
37. #Attributes
38. self.\_\_cd\_id = len(cd\_table) + 1
39. self.\_\_cd\_title = title
40. self.\_\_cd\_artist = artist
42. #Properties
43. @property
44. **def** cd\_id(self):
45. """getter for cd\_id
46. args: none
47. returns: cd\_id (int)
48. """
49. **return** self.\_\_cd\_id
51. @cd\_id.setter
52. **def** cd\_id(self, cd\_idValue):
53. """setter for cd\_id
54. args: cd\_idValue (int)
55. returns: none
56. """
57. **if** str(cd\_idValue).isnumeric() == False:
58. **raise** Exception('ID must be numeric') #This should not happen because ID number is set automatically, but wanted to build in an exception just in case
59. **else**:
60. self.\_\_cd\_id = cd\_idValue
62. @property
63. **def** cd\_title(self):
64. """getter for cd\_title
65. args: none
66. returns: cd\_title (str)
67. """
68. **return** self.\_\_cd\_title.title()
70. @cd\_title.setter
71. **def** cd\_title(self, cd\_titleValue):
72. """setter for cd\_title
73. args: cd\_titleValue (str)
74. returns: none
75. """
76. self.\_\_cd\_title = cd\_titleValue.title()
78. @property
79. **def** cd\_artist(self):
80. """getter for cd\_artist
81. args: none
82. returns: cd\_artist (str)
83. """
84. **return** self.\_\_cd\_artist
86. @cd\_artist.setter
87. **def** cd\_artist(self, cd\_artistValue):
88. """setter for cd\_artist
89. args: cd\_artistValue (str)
90. returns: none
91. """
92. self.\_\_cd\_artist = cd\_artistValue
93. #Methods
94. **def** \_\_str\_\_(self):
95. """returns formatted string of """
96. **return** 'ID #{}: {} (by: {})'.format(self.cd\_id, self.cd\_title, self.cd\_artist)
98. # -- PROCESSING -- #
99. **class** FileIO:
100. """
101. Processes data to and from file:
102. properties:
104. methods:
105. save\_file(file\_name, cd\_table): -> None
106. read\_file(file\_name, cd\_table): -> (a list of CD objects)
107. """
108. # TODO Add code to process data from a file
109. # TODO Add code to process data to a file
110. @staticmethod
111. **def** read\_file(file\_name, cd\_table):
112. """Function to manage data ingestion from file to a list of dictionaries
113. Reads the data from file identified by file\_name into a 2D table
114. (list of dicts) 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. cd\_table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
118. Returns:
119. cd\_table (list of dict): 2D data structure read from the binary file.
120. """
121. **try**:
122. cd\_table.clear()  # this clears existing data and allows to load data from file
123. with open(file\_name, 'rb') as objFile:
124. cd\_table = pickle.load(objFile)
125. **print**('Data successfully loaded from file.')
126. **return** cd\_table
127. **except** FileNotFoundError:
128. **print**(file\_name, 'was not found.')
129. **return** cd\_table #must return this so that function returns a list, cannot assign None to a list type variable
130. **except** EOFError:
131. **print**(file\_name, 'is empty.')
132. **return** cd\_table
134. @staticmethod
135. **def** save\_file(file\_name, cd\_table):
136. """
137. Function to save table to a file. Includes error handling.
138. Args: cd\_table (list): name of list used to store dictionaries
139. file\_name (string): the name of the .txt file where data will be written.
140. Returns: none
141. """
142. **try**:
143. with open(file\_name, 'wb') as objFile:
144. pickle.dump(cd\_table, objFile)
145. **print**('Data successfully saved to file.')
146. **except**:
147. **print**('Error encountered while saving. Please try again.')
149. **class** DataProcessor:
151. @staticmethod
152. **def** add\_cd(new\_cd, cd\_table):
153. """
154. Creates dictionary from data received in IO.user\_cd\_input. Appends dictionary to cd\_table.
155. Args: cd\_table (list): name of list used to store dictionaries
156. new\_cd (CD object):
157. Returns: none
159. try:
160. for row in cd\_table:
161. if row['ID'] == intID:
162. raise DuplicateIDError()
163. cd\_dict = {'ID': intID, 'Title': strTitle, 'Artist': strArtist}
164. cd\_table.append(cd\_dict)
165. except DuplicateIDError as e:
166. print('You entered a duplicate ID number, therefore your CD entry was not added to the inventory.\nPlease enter a unique ID number.\n')
167. print (e.\_\_str\_\_)
168. """
169. cd\_table.append(new\_cd)
171. @staticmethod
172. **def** delete\_cd(cd\_table, IDnumber):
173. """
174. Function to delete a CD's dictionary based on user's choice of which ID number to delete.
175. Loops through the 'ID' key's value in each dictionary and checks if it is equal to the user's input.
176. Renumbers all ID numbers after deletion.
177. Args: table (list): name of list used to store dictionaries
178. IDnumber (int): user's input of which ID to delete
179. Returns: none
180. """
181. intRowNr = -1
182. blnCDRemoved = False
183. **for** row **in** cd\_table:
184. intRowNr += 1
185. **if** row.cd\_id == IDnumber:
186. **del** cd\_table[intRowNr]
187. blnCDRemoved = True
188. **break**
189. **if** blnCDRemoved:
190. **print**('The CD was removed')
191. **else**:
192. **print**('Could not find this CD!')
193. **for** i **in** range(len(cd\_table)): #These two lines renumber all ID numbers after deletion so that ID numbers are sequential
194. cd\_table[i].cd\_id = i + 1
196. # -- PRESENTATION (Input/Output) -- #
197. **class** IO:
198. # TODO add docstring
199. # TODO add code to show menu to user
200. @staticmethod
201. **def** print\_menu():
202. """Displays a menu of choices to the user
203. Args:
204. None.
205. Returns:
206. None.
207. """
208. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
209. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
210. # TODO add code to captures user's choice
211. @staticmethod
212. **def** menu\_choice():
213. """Gets user input for menu selection
214. Args:
215. None.
216. Returns:
217. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
218. """
219. choice = ' '
220. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
221. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
222. **print**()  # Add extra space for layout
223. **return** choice
224. # TODO add code to display the current data on screen
225. @staticmethod
226. **def** show\_inventory(cd\_table):
227. """Displays current inventory table
228. Args:
229. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
230. Returns:
231. None.
232. """
233. **print**('======= The Current Inventory: =======')
234. **for** row **in** cd\_table:
235. **print**(row.\_\_str\_\_()) #Simply prints formatted string returned by \_\_str\_\_ method
236. **print**('======================================')
238. # TODO add code to get CD data from user
239. @staticmethod
240. **def** get\_int\_input(input\_string):
241. """ Gets an integer input from the user
242. Args:
243. input\_string (string): String displayed prompting user for input
244. Returns:
245. user\_int\_input (int): Integer value given by the user
246. """
247. correctID = False
248. **while** correctID == False:
249. **try**:
250. user\_int\_input = int(input(input\_string).strip()) #Structured error handling for when ID is not an int
251. correctID = True
252. **return** user\_int\_input
253. **except** ValueError:
254. **print**('You entered an invalid ID. Please enter an integer.')
256. @staticmethod
257. **def** user\_cd\_input():
258. """Gathers user input for fields of the new CD
259. Args:
260. None.
261. Returns:
262. intID (int): ID for the new CD
263. strTitle (string): Title of the new CD
264. strArtist (string): Artist of the new CD
265. """
266. #intID = IO.get\_int\_input("Enter ID: ")  #The get\_int\_input() function contains error handling in case input is a non-integer
267. strTitle = input('What is the CD\'s title? ').strip()
268. strArtist = input('What is the Artist\'s name? ').strip()
270. **return** strTitle, strArtist

273. # -- Main Body of Script -- #
274. # TODO Add Code to the main body
275. # Load data from file into a list of CD objects on script start
276. **try**:    #Try to read in file, if it exists
277. lstOfCDObjects = FileIO.read\_file(strFileName, lstOfCDObjects) #loads file contents into memory, stored in lstOfCDObjects
278. **print**('Welcome back to your CD Inventory')
279. **except**:#If file does not exist, continue anyways
280. **print**('Welcome to your CD Inventory! An existing inventory file was not found.\nLet\'s create a new inventory.\n')
281. # Display menu to user
282. **while** True:
283. choice = IO.menu\_choice()
284. # show user current inventory
285. **if** choice == 'i':
286. IO.show\_inventory(lstOfCDObjects)
287. # let user add data to the inventory
288. **elif** choice == 'a':
289. new\_cd = CD(\*IO.user\_cd\_input(), lstOfCDObjects)
290. DataProcessor.add\_cd(new\_cd, lstOfCDObjects)
291. IO.show\_inventory(lstOfCDObjects)
292. # let user save inventory to file
293. **elif** choice == 's':
294. FileIO.save\_file(strFileName, lstOfCDObjects)
295. # let user load inventory from file
296. **elif** choice == 'l':
297. FileIO.read\_file(strFileName, lstOfCDObjects)
298. # let user delete CD from inventory
299. **elif** choice == 'd':
300. IO.show\_inventory(lstOfCDObjects)
301. intIDDel = IO.get\_int\_input('Enter the ID you would like to delete: ')
302. DataProcessor.delete\_cd(lstOfCDObjects, intIDDel)
303. IO.show\_inventory(lstOfCDObjects)
304. # let user exit program
305. **elif** choice == 'x':
306. **break**
307. **else**:
308. **print**('General error while making menu choice.')