John Hunter

March 2, 2020

Foundations of Programing: Python

Module 06

Knowledge Document

Introduction

The module this week continues on many topics that have been introduced previously. The discussion here on functions continues the separation of concerns discussion from last week. Parameters, Arguments, and Return values are all integral to passing data in and out of functions, and processing that data inside the function. The other topics help to either support the furthering of the SoC type programming practices, e.g. Overloaded Functions, or to give more flexibility to the function call, e.g. None, Variable Scope, and Classes. Doc String is also introduced, which helps to document the function so that the input, output, and processing can not only be read by any programmer, but so that the code base that may need to incorporate the function can do so easily, and in some cases, automatically.

Materials Discussion

Functions are widely used in most programing languages as a way to reuse the same piece of code without having to type, or update the same code in multiple places. By calling a function, the execution of which involves possibly hundreds of lines of code, the call can accomplish the same with only a few lines of code. The wide use of functions presents opportunities for standardization. Naming conventions, structure and Docstrings all are types of standardization that improve readability and ease of code integration. Some simple, best practices, e.g. naming your function something useful, for instance calculateValue, in which we have stated the purpose of the function and used a case sensitivity pattern to eliminate any remaining doubt. Case sensitivity patterns, like camelCase, or UpperCamelCase, can be used at the organization level, or at the application level to specify variable names, function names, or both or some other programming convention as desired by the developer.

Function calls typically pass data in and out. If a function serves a purpose, then there are typically parameters of the function, which will need to represent the value of a variable, at the point where the function is called. It is rarely useful for a function parameter to be a constant. A developer could at any point in the execution define a constant and use it. Parameters then, generally are variables passed to the function to calculate some value. That calculated value is typically the return of the function. This course has seen many data types be used to define variables, and they are all available to be passed to and returned from a function. It can be useful to pass a value as a return. However, it is generally assigned to a variable so that the value can be reused as desired. While multiple values, unlimited in fact, can be passed to the function upon a call, the return is a single variable. The return can be a package of many values, but would have to be parsed outside the function if needed.

In mathematics it is generally forbidden to overload a variable. Python is built to support overloading of functions. The intended purpose is to define different functionality depending on the parameter set that is passed. We can then define a function func(X,Y), such that func(X,Y) =! func(Z, J, K). In one case func(X,Y) might return a list, where as func(Z,J,K) might return a null variable, also called by the keyword ‘none’. In this case, depending on the construction of the function, the entire purpose of the function could be different, or we could have the fact that only two variables are available, dictate that the two-value version be used based on some conditional statement.

References can be used to point to a variable location instead of the value. A basic example is used where a variable x is assigned a list of values (a,b), another variable is assigned to refer to x, as y=x, the first position of x is change, so that x=(c,b), and y is printed as (c,b). In this example, y is never assigned the value (c,b), but it is passed (c,b) by reference.

Doc String is manner in which functions are documented with a header like comment that describes the functionality enabled. The reasons docstrings are used are many, with readability, maintainability, and consistency all being principle selling points of this type of standardization.

Doc String focuses code review on overall function of the application instead of one module through ease of understanding. A developer could use the docstrings of all the modules in an application to automate reports on the over all performance. Docstrings are string literals that occur in the first line of the body of a definition of a class, function, module, or method.

Classes are typically thought of as an object type definition. In addition to defining custom object types, classes can be thought of as combining the custom type definition with functionality. Object Oriented Programming in other languages is dependent on custom class definitions. The overloading of class definitions in Python allows for complex inheritance hierarchies from base classes and layered data reuse.

Summary

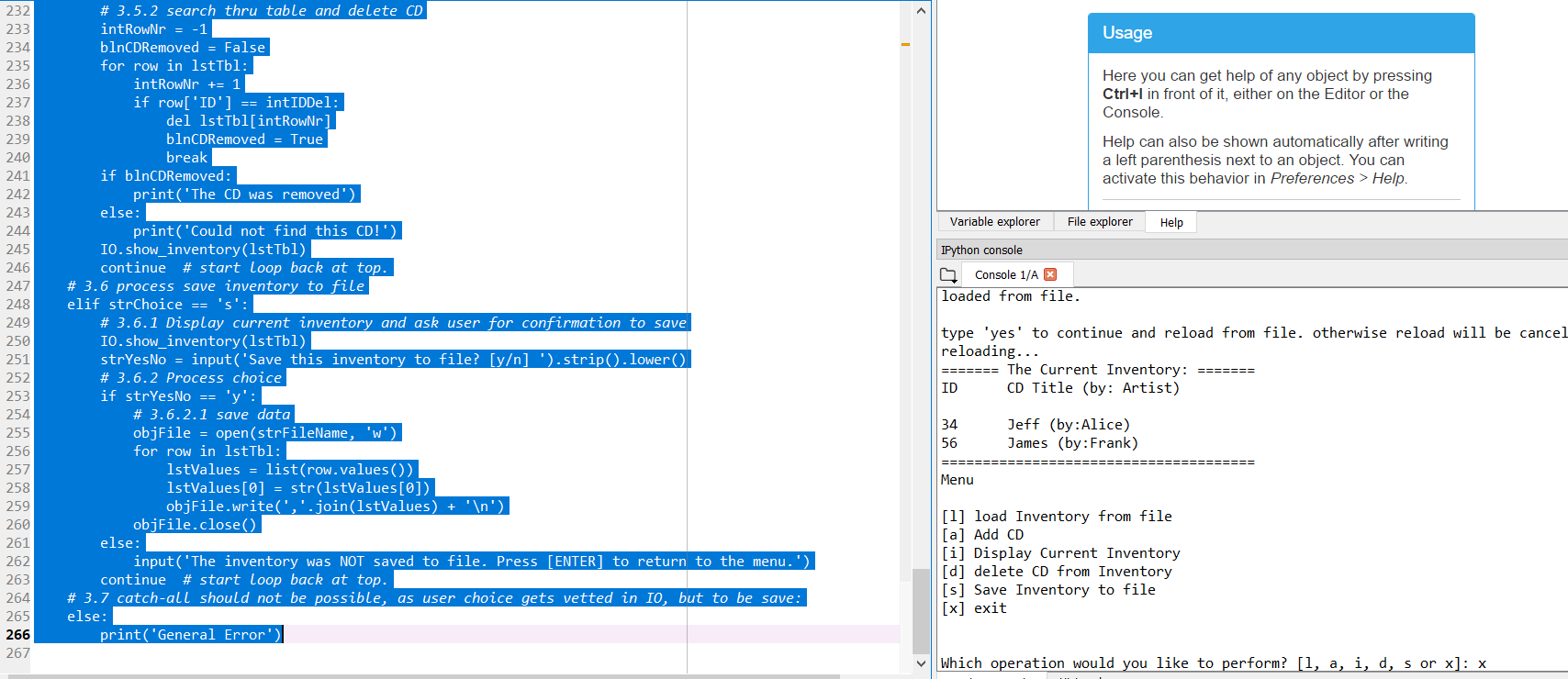


Figure Spyder runs the code. The majority of the functionality should be the same as the previous assignment.

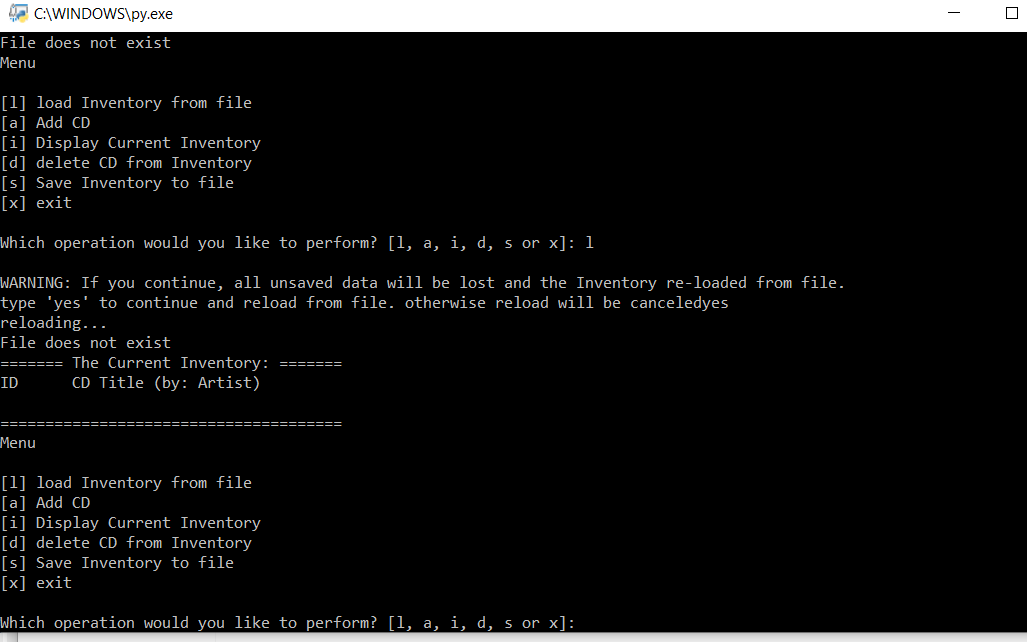


Figure The command prompt executes the script as expected.

There are many was in which the application can be structures to fit the need of the user or developer. Functions provide easy data management while allowing for the reuse of code and the standardization of documentation, name spaces. Although we have not employed some aspects of the lesson, e.g. overloaded functions, we could have done. Standardization is key when your code could be reviewed by other developers who have no understanding of the code, or function it is meant to perform. Classes, docstring, functions, and case sensitivity all help the modern developer manage variables, document code, and provide functionality in an easily readable and integratable format.

Appendix

Planet B Syntax Highlighter

1. #----------------------------------------#
2. # Assignment06: CDInventory.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 functions.
6. # Change Log:
7. # Johnh, 2020-Feb-26 Script Created from starter assignment text .py file
8. # Johnh, 2020-Feb-26 Header and script structure updated
9. # Johnh, 2020-Feb-27 Updated code from previous assignment
10. # Johnh, 2020-Mar-1  Added functiona and static methods
11. # Johnh, 2020-Mar-2 Bug fixed, code running
12. # Johnh, 2020-Mar-2 Readablity Review, comments cleanup, SoC formatting
13. # Johnh, 2020-Mar-2 Tested, passed
14. # Johnh, 2020-Mar-3 Added to Git and submitted
15. #----------------------------------------#
17. **import** os.path
18. # Corrects for a bug in the original file whereby the script did not run
19. # if there was not already .txt file
20. **from** os **import** path
22. # -- DATA -- #
23. strChoice = '' # User input
24. lstTbl = []  # list of lists to hold data
25. dicRow = {}  # list of data row
26. strFileName = 'CDInventory.txt'  # data storage file
27. objFile = None  # file object
29. # -- PROCESSING -- #
30. **class** DataProcessor:
31. @staticmethod
32. **def** delete\_cd(intIDDel,table):
33. """Deletes a CD row from the table
35. Args:
36. intIDDel (int): ID indicates user entry to delete
37. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
39. Returns:
40. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
41. """
42. intRowNr = -1
43. blnCDRemoved = False
44. **for** row **in** table:
45. intRowNr += 1
46. **if** row['ID'] == intIDDel:
47. **del** table[intRowNr]
48. blnCDRemoved = True
49. **break**
50. **if** blnCDRemoved:
51. **print**('The CD was removed')
52. **else**:
53. **print**('Could not find this CD!')
54. **return** table
56. @staticmethod
57. **def** ask():
58. """Ask user for new ID, CD Title and Artist
60. Args:
61. None
63. Returns:
64. dicRow (dictionary):  A dictionary entry with ID (int): integer holds
65. the ID tag,title (string): string holds the name of the CD
66. and an artist (string): string holds the name of the Artist.
67. """
68. strID = input('Enter ID: ').strip()
69. strID = int(strID)
70. strTitle = input('What is the CD\'s title? ').strip()
71. stArtist = input('What is the Artist\'s name? ').strip()
72. dicRow = {'ID': strID, 'CD Title': strTitle, 'Artist': stArtist}
74. **return** dicRow
75. **pass**
77. **class** FileProcessor:
78. """Processing the data to and from text file"""
80. @staticmethod
81. **def** read\_file(file\_name, table):
82. """Function to manage data from file to a list of dictionaries
84. Reads the data from file file\_name into a 2D table
85. (list of dicts) table one line in the file represents one dictionary row in table.
87. Args:
88. file\_name (string): name of file used to read the data from
89. table (list of dict): 2D data structure (list of dicts) holds the data during runtime
91. Returns:
92. None.
93. """
94. **if** path.exists('CDInventory.txt'):
95. table.clear()  # this clears existing data and allows to load data from file
96. objFile = open(file\_name, 'r')
97. **for** line **in** objFile:
98. data = line.strip().split(',')
99. dicRow = {'ID': int(data[0]), 'Title': data[1], 'Artist': data[2]}
100. table.append(dicRow)
101. objFile.close()
102. **else**:
103. **print** ("File does not exist")
105. @staticmethod
106. **def** write\_file(file\_name, table):
107. """Writes the inventory of IDs, CD Names, and Artists to a text file
109. Args:
110. file\_name (string): The name of the file that it will write to
111. table (list of dict): 2D data structure (list of dicts) holds the data during runtime
113. Returns:
114. None but saves a file in the directory of the python script
116. """
118. objFile = open(file\_name, 'w')
119. **for** row **in** table:
120. strRow = ''
121. **for** item **in** row.values():
122. strRow += str(item) + ','
123. strRow = strRow[:-1] + '\n'
124. objFile.write(strRow)
125. objFile.close()
126. **pass**

129. # -- PRESENTATION (Input/Output) -- #
131. **class** IO:
132. """Handling Input / Output"""
134. @staticmethod
135. **def** print\_menu():
136. """Displays a menu of choices to the user
138. Args:
139. None.
141. Returns:
142. None.
143. """
145. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
146. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
148. @staticmethod
149. **def** menu\_choice():
150. """Gets user input for menu selection
152. Args:
153. None.
155. Returns:
156. choice (string): a lower case string of the users input l, a, i, d, s or x
158. """
159. choice = ' '
160. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
161. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
162. **print**()  # Add extra space for layout
163. **return** choice
165. @staticmethod
166. **def** show\_inventory(table):
167. """Displays current inventory table

170. Args:
171. table (list of dict): 2D data structure (list of dicts) holds the data during runtime.
173. Returns:
174. None.
176. """
177. **print**('======= The Current Inventory: =======')
178. **print**('ID\tCD Title (by: Artist)\n')
179. **for** row **in** table:
180. **print**('{}\t{} (by:{})'.format(\*row.values()))
181. **print**('======================================')
183. # 1. When program starts, read in the currently saved Inventory
184. FileProcessor.read\_file(strFileName, lstTbl)
186. # 2. start main loop
187. **while** True:
188. # 2.1 Display Menu to user and get choice
189. IO.print\_menu()
190. strChoice = IO.menu\_choice()
192. # 3. Process menu selection
193. # 3.1 process exit first
194. **if** strChoice == 'x':
195. **break**
196. # 3.2 process load inventory
197. **if** strChoice == 'l':
198. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
199. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled')
200. **if** strYesNo.lower() == 'yes':
201. **print**('reloading...')
202. FileProcessor.read\_file(strFileName, lstTbl)
203. IO.show\_inventory(lstTbl)
204. **else**:
205. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
206. IO.show\_inventory(lstTbl)
207. **continue**  # start loop back at top.
208. # 3.3 process add a CD
209. **elif** strChoice == 'a':
210. # 3.3.1 Ask user for new ID, CD Title and Artist
211. strID = input('Enter ID: ').strip()
212. strTitle = input('What is the CD\'s title? ').strip()
213. stArtist = input('What is the Artist\'s name? ').strip()
215. # 3.3.2 Add item to the table
216. intID = int(strID)
217. dicRow = {'ID': intID, 'Title': strTitle, 'Artist': stArtist}
218. lstTbl.append(dicRow)
219. IO.show\_inventory(lstTbl)
220. **continue**  # start loop back at top.
221. # 3.4 process display current inventory
222. **elif** strChoice == 'i':
223. IO.show\_inventory(lstTbl)
224. **continue**  # start loop back at top.
225. # 3.5 process delete a CD
226. **elif** strChoice == 'd':
227. # 3.5.1 get Userinput for which CD to delete
228. # 3.5.1.1 display Inventory to user
229. IO.show\_inventory(lstTbl)
230. # 3.5.1.2 ask user which ID to remove
231. intIDDel = int(input('Which ID would you like to delete? ').strip())
232. # 3.5.2 search thru table and delete CD
233. intRowNr = -1
234. blnCDRemoved = False
235. **for** row **in** lstTbl:
236. intRowNr += 1
237. **if** row['ID'] == intIDDel:
238. **del** lstTbl[intRowNr]
239. blnCDRemoved = True
240. **break**
241. **if** blnCDRemoved:
242. **print**('The CD was removed')
243. **else**:
244. **print**('Could not find this CD!')
245. IO.show\_inventory(lstTbl)
246. **continue**  # start loop back at top.
247. # 3.6 process save inventory to file
248. **elif** strChoice == 's':
249. # 3.6.1 Display current inventory and ask user for confirmation to save
250. IO.show\_inventory(lstTbl)
251. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
252. # 3.6.2 Process choice
253. **if** strYesNo == 'y':
254. # 3.6.2.1 save data
255. objFile = open(strFileName, 'w')
256. **for** row **in** lstTbl:
257. lstValues = list(row.values())
258. lstValues[0] = str(lstValues[0])
259. objFile.write(','.join(lstValues) + '\n')
260. objFile.close()
261. **else**:
262. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
263. **continue**  # start loop back at top.
264. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be save:
265. **else**:
266. **print**('General Error')

References

https://youtu.be/ZCIa2i\_TfyE

https://youtu.be/4fR0zdAZ9QU

https://youtu.be/QhgN1k\_iJzI

https://youtu.be/U9wRmnWesRk

https://youtu.be/o5JExa35hQ0

https://www.learnpython.org/en/Functions

https://youtu.be/\_ypAw\_pCOt8

Michael Dawson Python Programming