Maria Dacutanan

Aug 20, 2020

Foundations of Programming: Python

Assignment 06

Functions and Classes

In this brand new module, I delved into the world of functions, it’s importance and how to define and call functions as well as how to group functions through classes. I learned how to pass values to function calls called arguments and how to define parameters and default values for these parameters. I gained understanding of variable scoping, global and local variables and how these variables behave inside an outside of functions.

I also revisited the importance of shared references but this time when using functions.

I learned about how to add documentation to the functions that I create myself as well as industry standards such as variable naming conventions

Functions

A function is a block of reusable code that performs a specific task. A function’s made advantage is that it helps break up code into smaller and modular chunks. As programs grow bigger, functions make code management easier. Furthermore, it keeps code looking clean and organized by avoiding code repetition and makes code reusable and shareable with other programmers.

Reusable Code

I have already mentioned some of the advantages of functions but in a bigger picture, here are a few important reasons for software reuse worth mentioning[[1]](#footnote-1):

* Increase company productivity
* Improve software quality
* Provide consistency across software products
* Improve Software performance

One way to reuse functions is to reuse them in your other programs or better yet, learn how to create modules so you can import functions into any program.

Abstraction in Object Oriented Programming

I must admit, I had to pause for a little bit to understand the concept of **abstraction** in the world of computer science and programming. This [website](https://stackify.com/oop-concept-abstraction/)[[2]](#footnote-2) explains it very well for a beginner such as me. I think for beginners in the programming world, especially if you intend to be a serious one, it is important to understand key concepts and principles such as this. It makes you a better programmer. Per the website I mentioned above, an abstraction’s main goal is to handle complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

In the real world, abstraction also applies. For example, making coffee with a coffee machine. You need to know how to use the coffee machine to make coffee. You provide water, coffee beans and switch the machine on. The thing you don’t know and don’t need to know is how the coffee machine works internally to brew the coffee. Applying this same concept in OOP, all you need to know are which methods are available for the objects and which input parameters are needed to trigger a specific operation. You don’t need to know how the method works internally.

Defining Functions

Basic syntax for defining function is pretty simple. You type the keyword **def** followed by a function name , followed by parenthesis () and then finally a colon. The statement for that function is indented. Figure 1 - Defining A Function, is an example of defining a function. The figure does include parameters which is explained a little bit later.

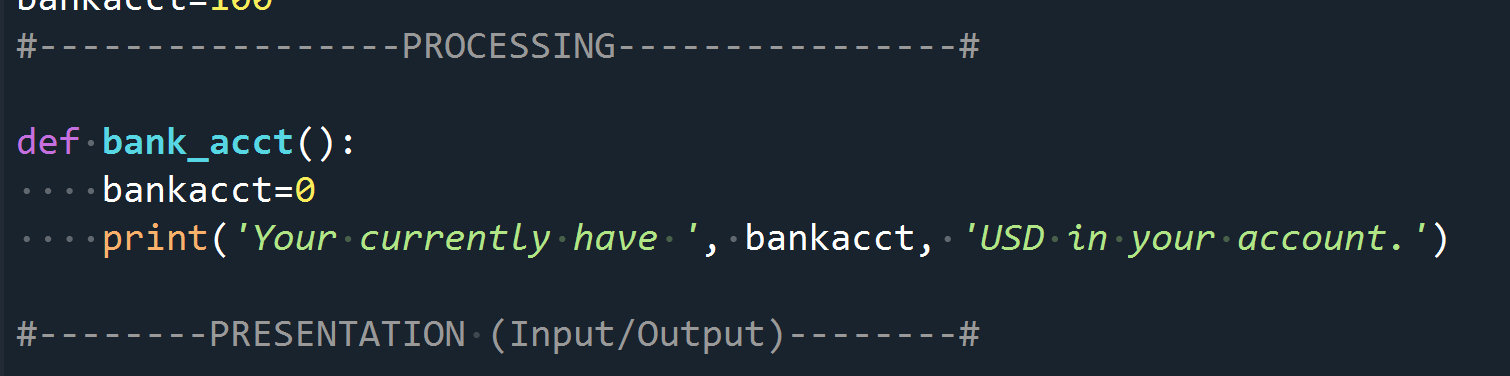


Figure - Defining A Function

Calling A Function

Basic syntax for calling function is also straightforward. You specify the function name followed by parenthesis(). In Figure 2 - Calling A Function, I simply typed in bankacct() to call the function.

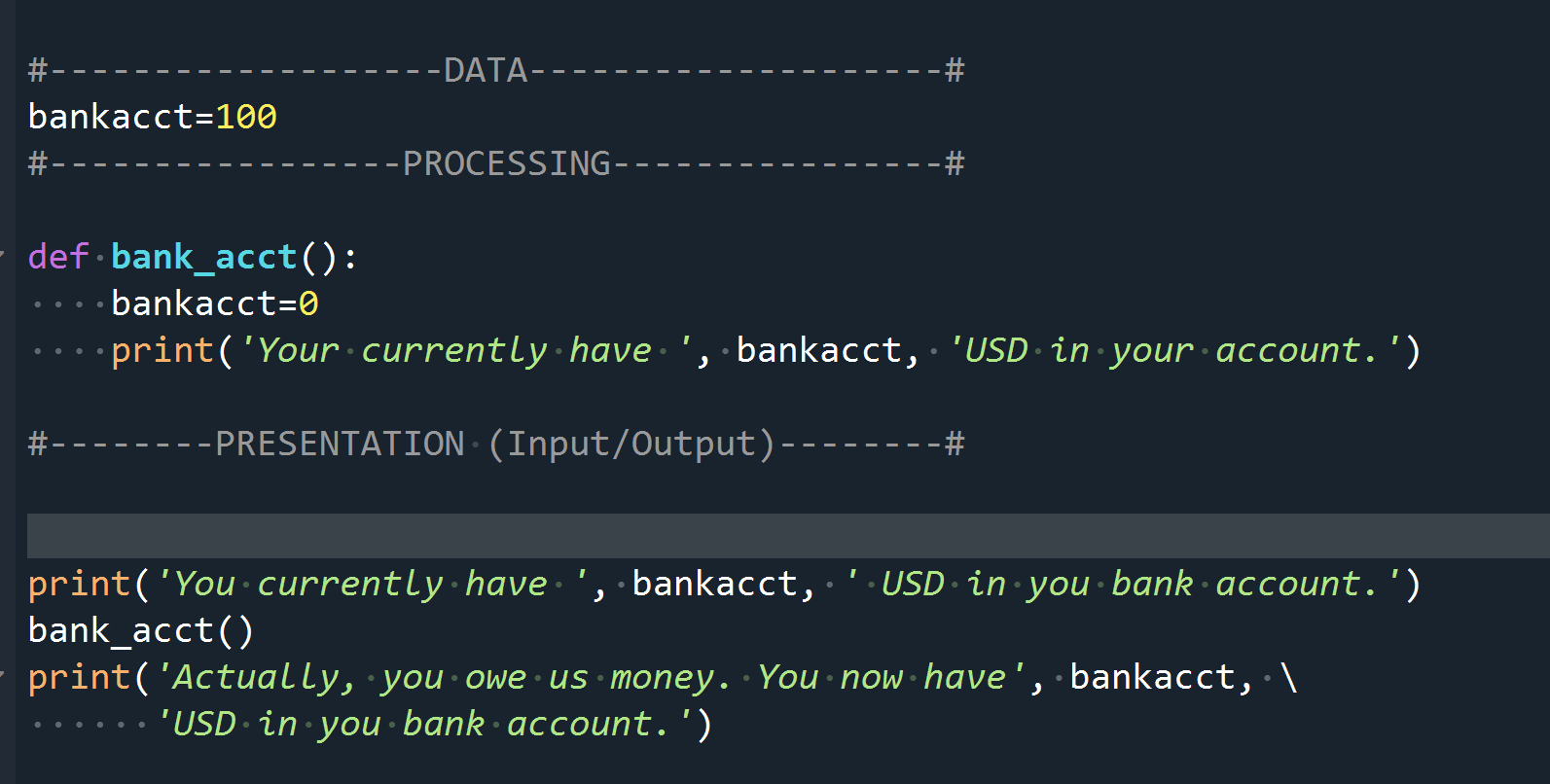
****

Figure - Calling A Function

Parameters and Arguments

In the section for defining functions, the basic syntax does not include **parameters**. Parameters are a way of catching values passed to the function which it uses to during the function block execution. Parameters are essentially variables defined within a function that can hold the **arguments** passed on during a function call. In other words, **parameters** are defined inside a function  while arguments are used during function calls, i.e. the values passed to function at run time.

When defining a function which includes parameters, basic syntax is as follows:

**def <function name>(<parameter>):**

**<statement(s)>**

You can of course define more than one parameter. Just need to list it out the parameters inside the parenthesis and comma-separate them.

In Figure 1 - Calling a Function With An Argument, I show how to call a function while passing an argument as well as defining a function that has a parameter definition. Figure 2 - Output of Code in Figure 1 is the result of the code in Figure 1.

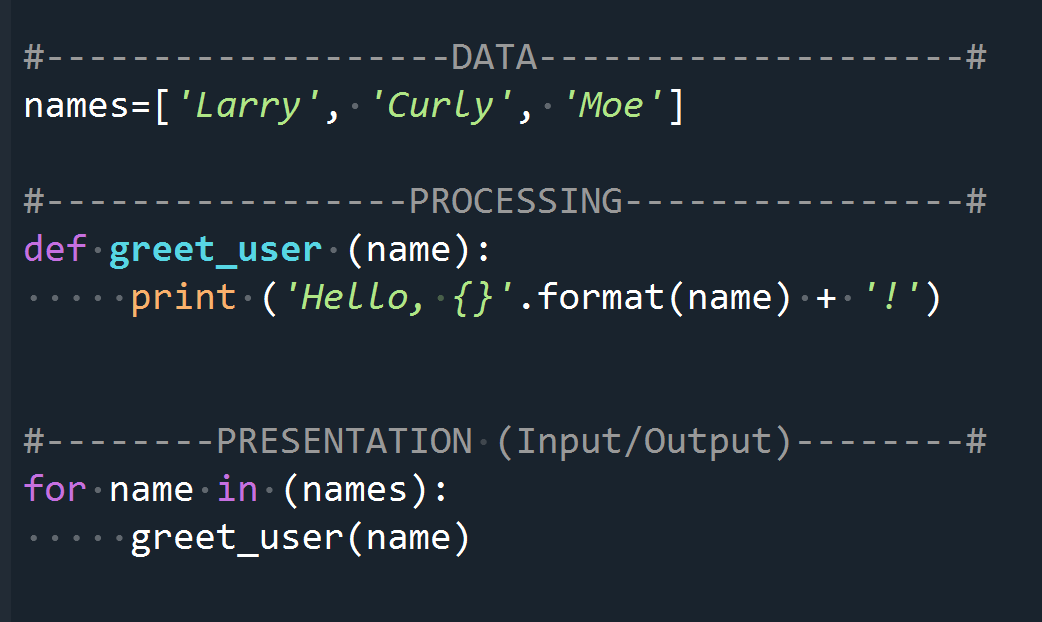


Figure - Calling a Function With An Argument

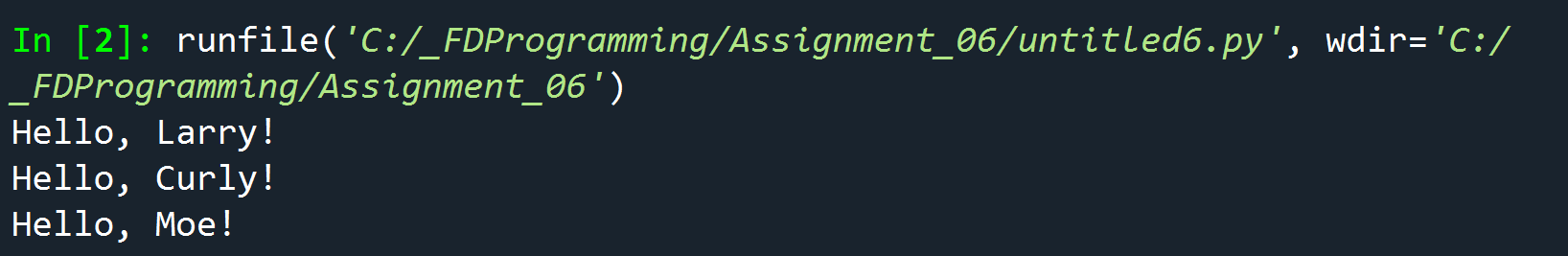


Figure - Output of Code in Figure 1

Positional Parameters and Positional Arguments

When a function is called with a series of arguments, you create **positional arguments**. Using **positional arguments** and **positional parameters** means that parameters get their values solely based on the position the values are sent, the first parameter gets the value of the first argument, the second parameters gets the value of the second argument sent, and so on.

Take a look at Figure 3 - Code with Positional Arguments and Positional Parameters. Function country\_capital has two defined parameters, val1 and val2. Now when I made a call to function country\_capital, I passed on two arguments: country and capital variables. When the function block is executed, the output is shown in Figure 4 - Output of Code in Figure 3

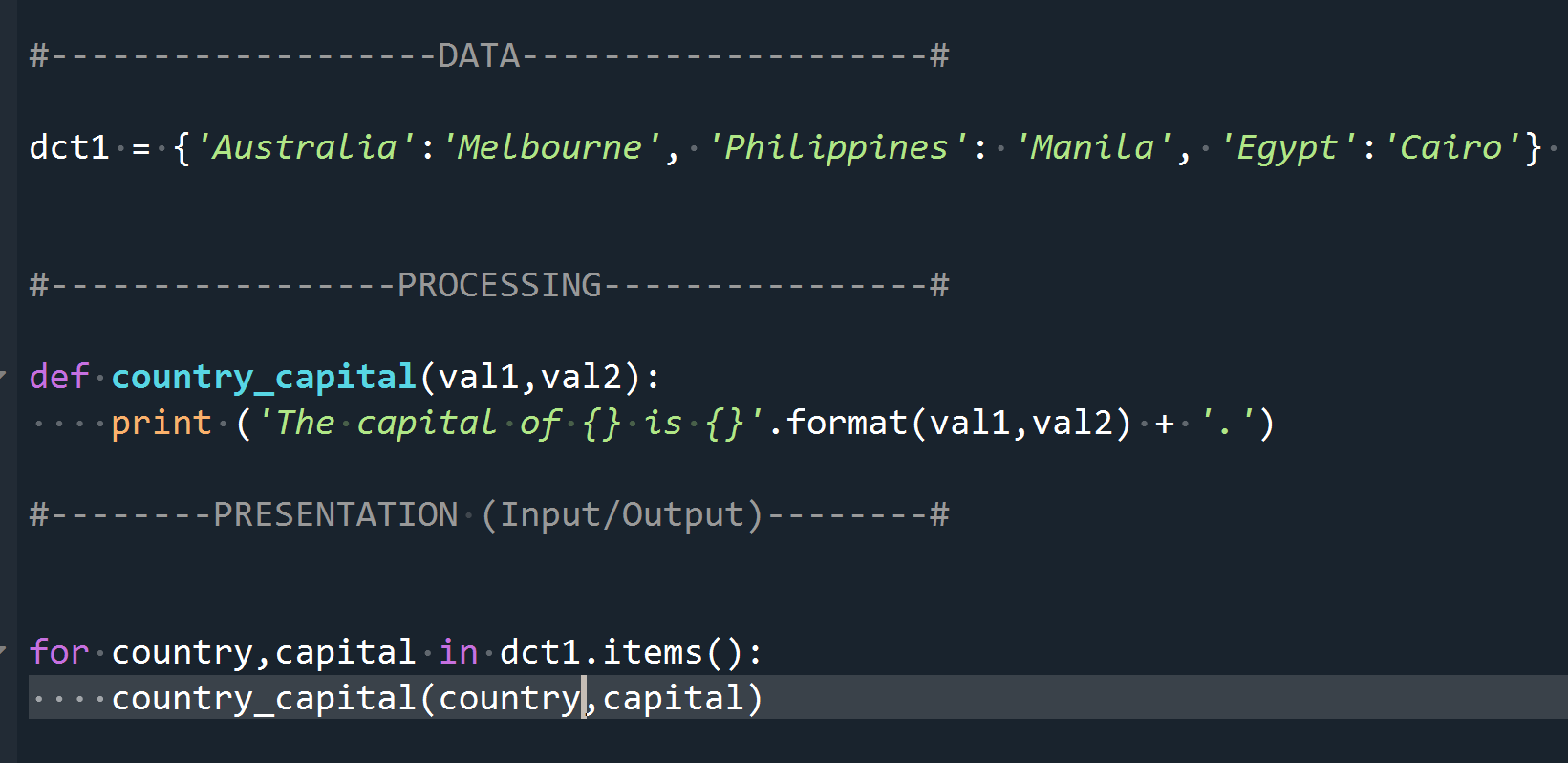
****

Figure - Code with Positional Arguments and Positional Parameters

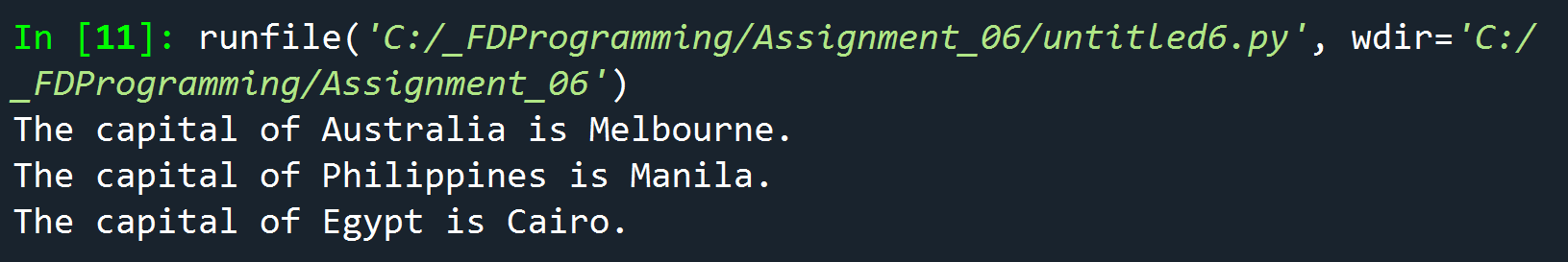


Figure - Output of Code in Figure 3

Now if during the function call I changed the way I switched the order of the arguments from country\_capital(country, capital) to country\_capital(capital\_country), the output would look like Figure 5 - Output of Argument Switch. This is because the order by which I sent the arguments during the function call does not switch automatically switch the order by which the parameters catch the values.

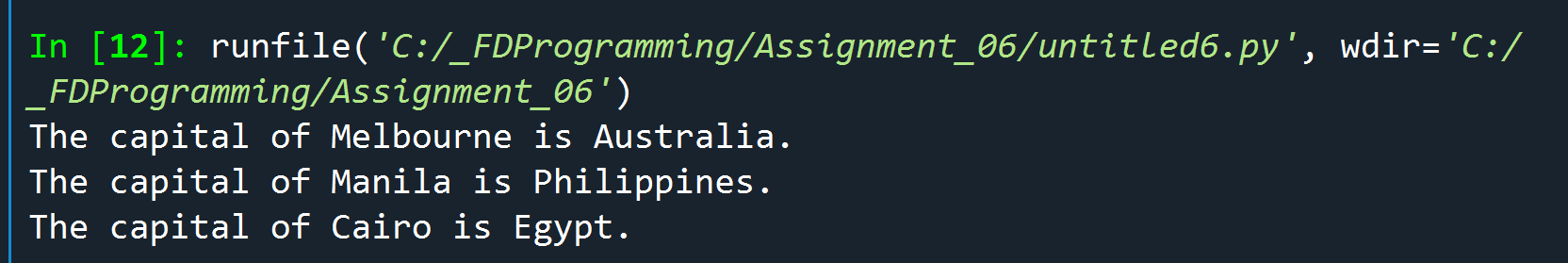


Figure - Output of Argument Switch

Keyword Arguments

You can tell the function to assign certain values to specific parameters, regardless of order, if you use **keyword arguments**. With keyword arguments, the order does not matter because you use the actual parameter names defined in the function header.

In Figure 6 - Keyword Arguments, when I called function country\_capital, instead of just simply passing on capital and country variables, I assigned these variables to the parameter names defined in the function header.

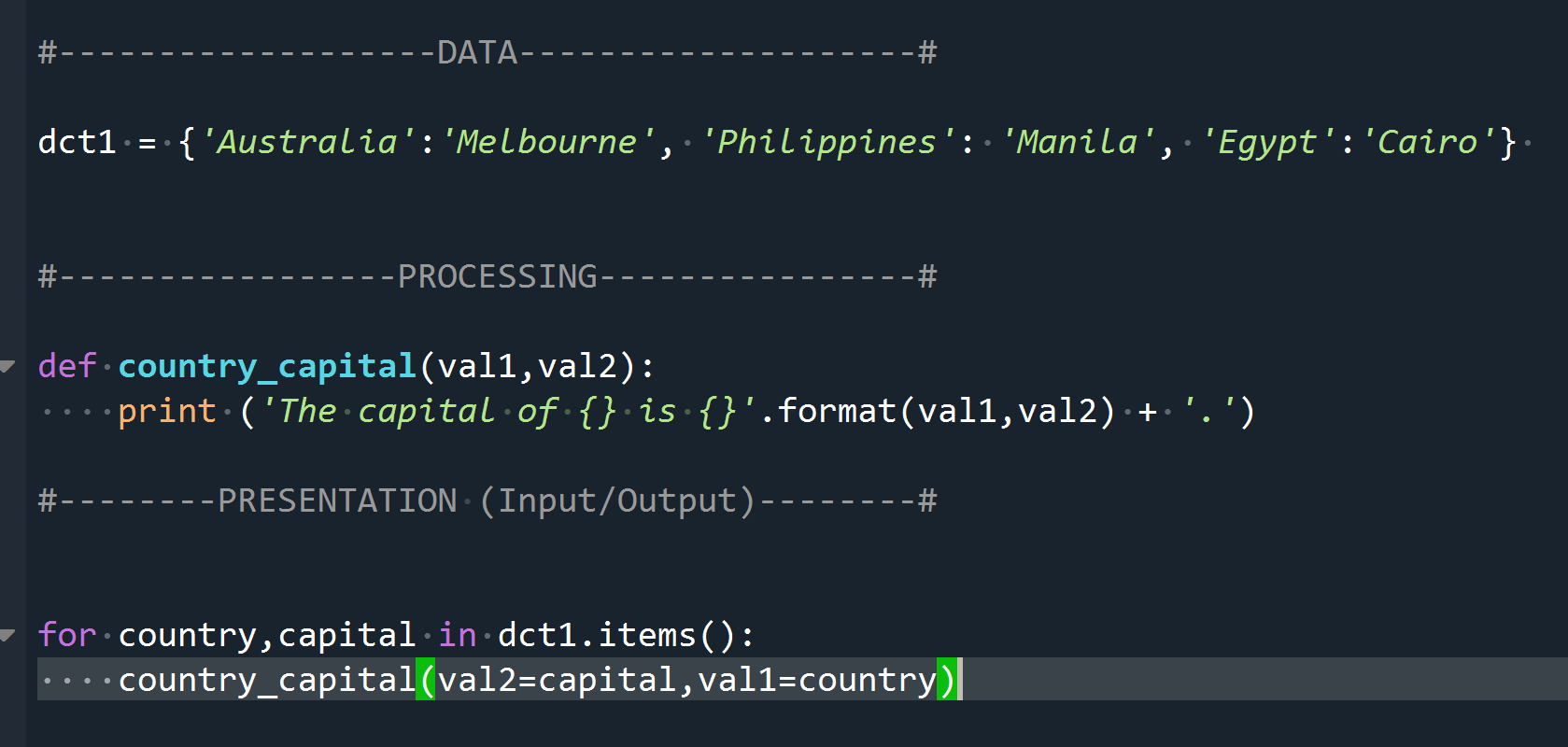


Figure – Keyword Arguments

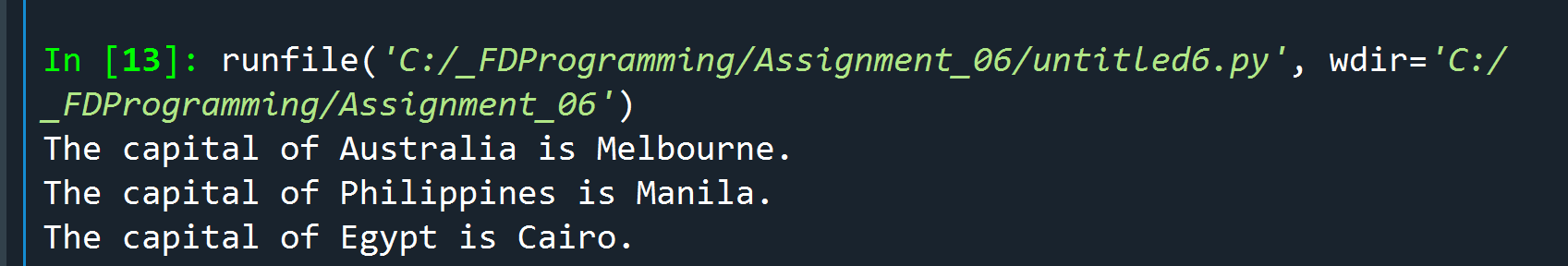


Figure – Code Output of Figure 6

As you can observe in Figure 7 – Code Output of Figure 6, even though I passed val2 first before val1 during the function call, the values are still caught and assigned to the right parameters because I have used the keyword (parameter) names as arguments.

Default Parameter Values

In cases where an argument is not passed on during the function call, default values can be assigned to the parameters. In Figure 8 - Default Parameter Values, I wrote a simple function that accepts two parameters, ‘lang’ and ‘name’. ‘lang’ variable has a default value of ‘English’ while ‘name’ has a default value of ‘user’. If the greeting function is called and either or both parameters did not receive arguments, the default values are used instead. Figure 9 - Code output of Figure 8 shows the output of the different scenarios where default parameters are used. The only caveat when defining default values for parameters, once you define a default value for a parameter, all succeeding parameters should have default values as well.



Figure - Default Parameter Values

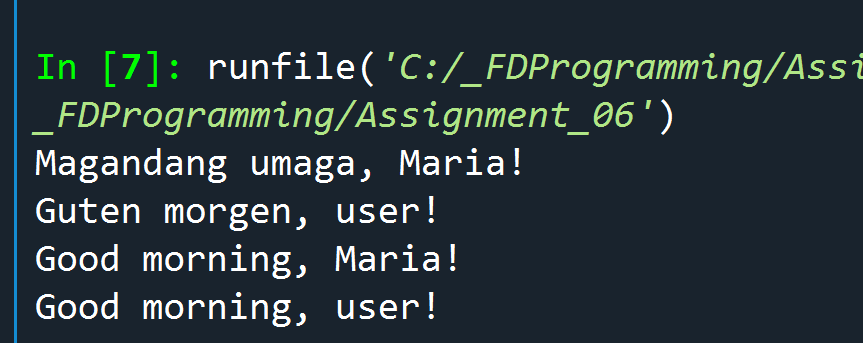


Figure - Code output of Figure 8

Return Values

A function can take in values and but it can also return value(s) back to the calling program. In our homework, I had to create a function that takes user input and then pass these values back to the main program. To return a value(s), you type the keyword **return** followed by the value that needs to be returned. If you have multiple values, just comma separate them. Figure 12 - Return Values is snippet of the get\_newInventory() function from CDInventory.py program and it illustrates the function returning 3 values back.

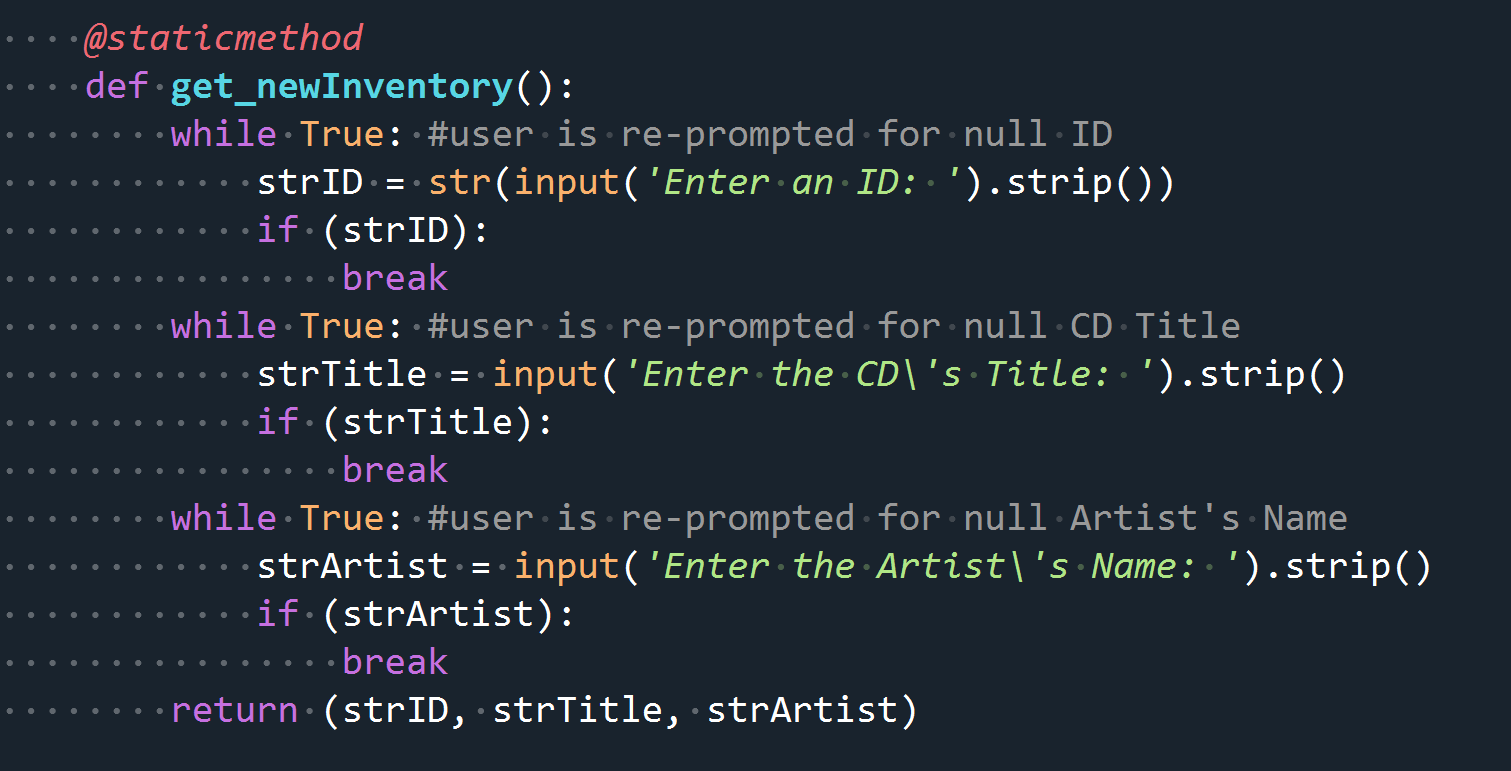


Figure - Return Values

Variable Scopes

Each function you define has its own **scope** – meaning it is self-contained and independent of the other functions in the program. Variables defined inside functions are called **local variables** and cannot be accessed outside of the function. Even if I had two variables of the same name being used by two separate functions, changes made to the variable in one function does not affect the other.

Any variables defined outside of functions are in a global scope and are therefore **global variables.**

Global Variables and Functions

A global variable can be read within any function but a function cannot change a global variable directly unless you use the **global** function. Figure 10 - Changing global variable inside a function demonstrates how to use the global function to change global variables within functions. Figure 11 - Code output of Figure 10 shows the output.

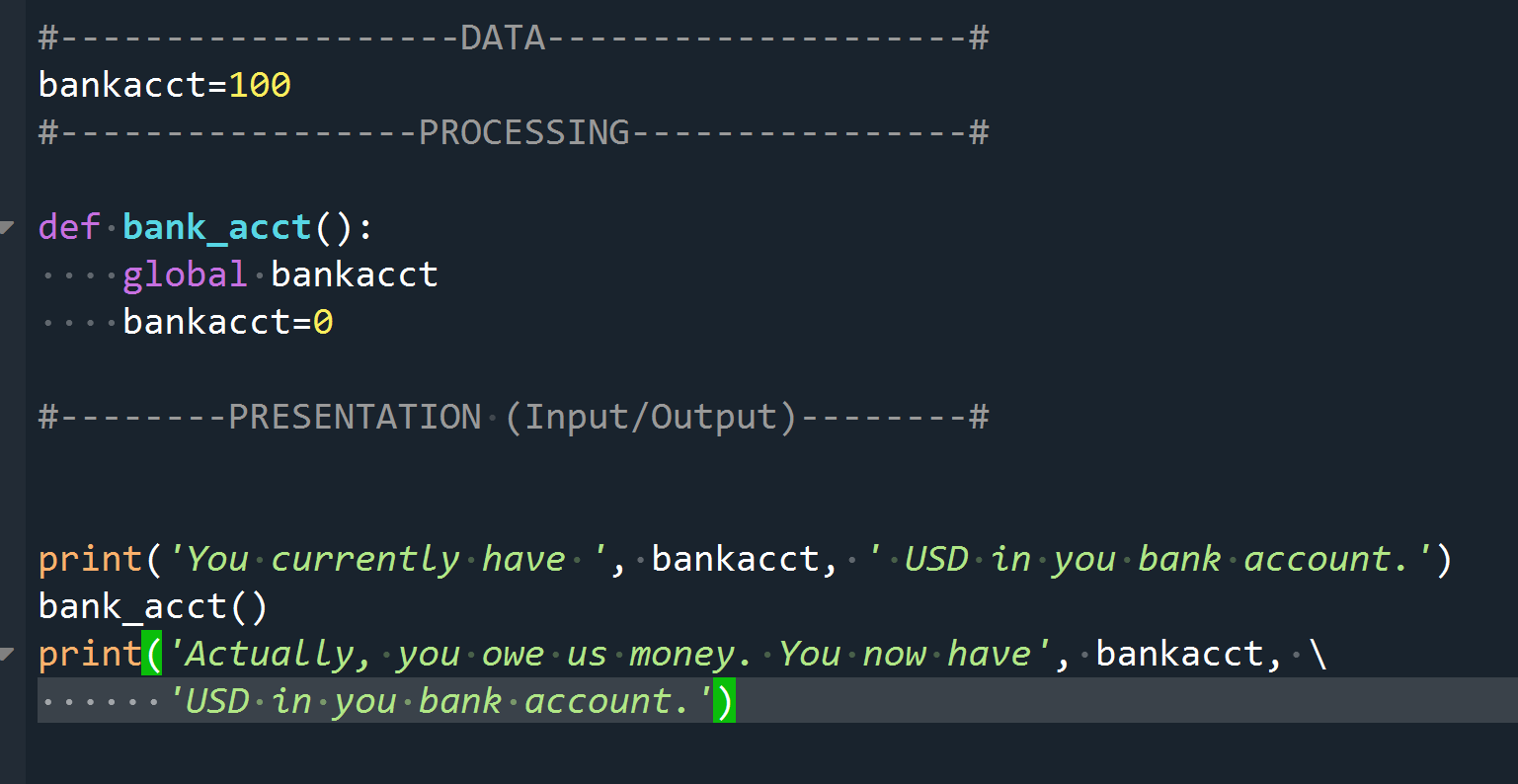


Figure - Changing global variable inside a function

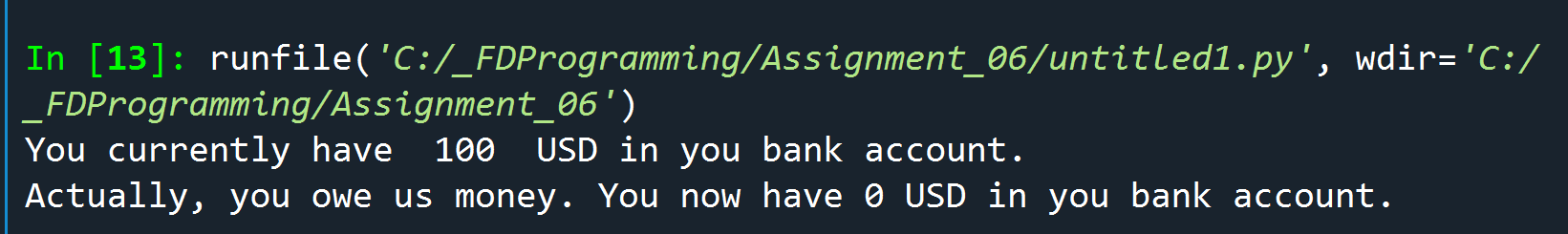


Figure - Code output of Figure 10

Shadowing Global Variables

When you create a variable that has the same name as global variable, you actually have created a **shadow** variable. Although it looks like the global variable, whatever you do to it inside the function actually does not affect the global variables. Please take a look at Figure 12 - Shadow Variable and the output it produced in Figure 13 - Code output of Figure 12

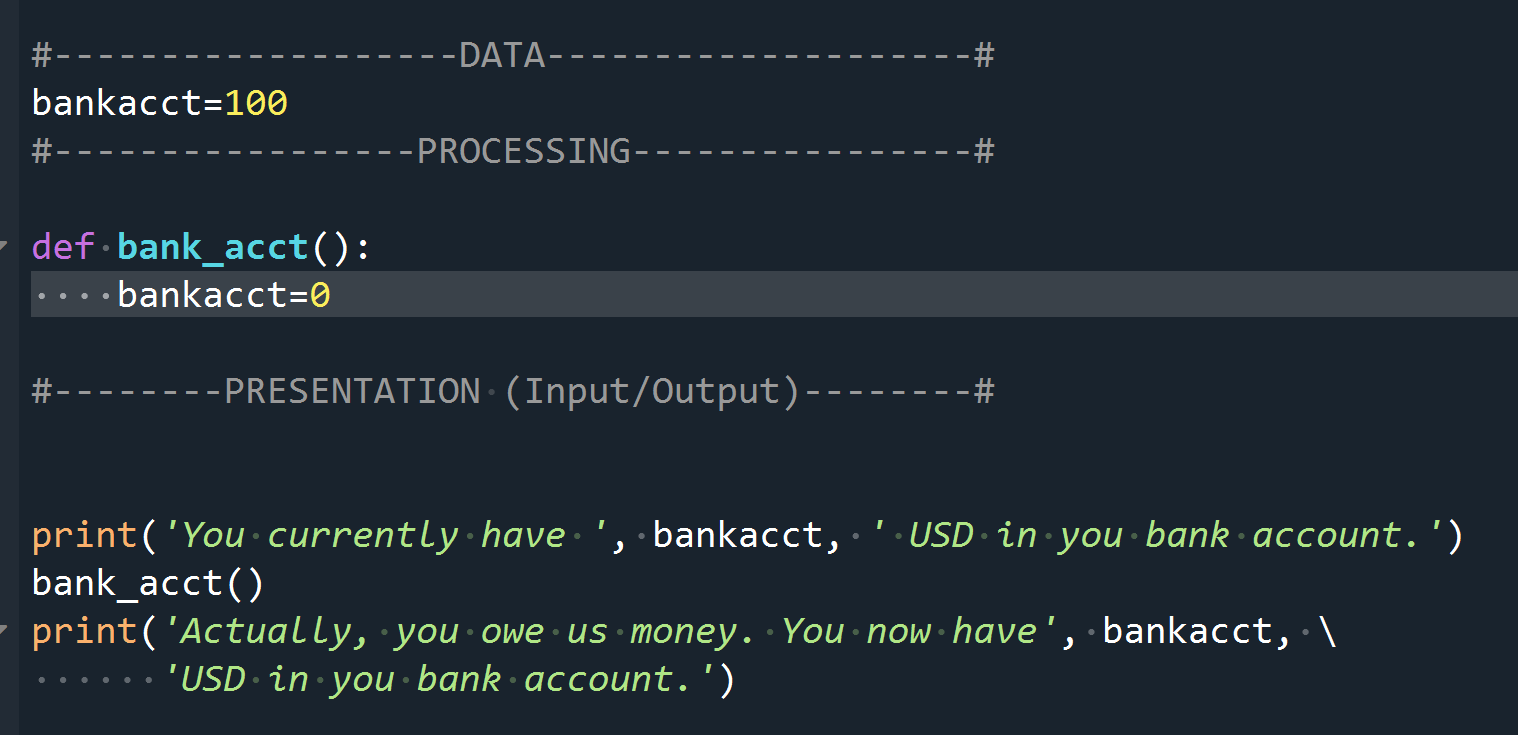


Figure - Shadow Variable

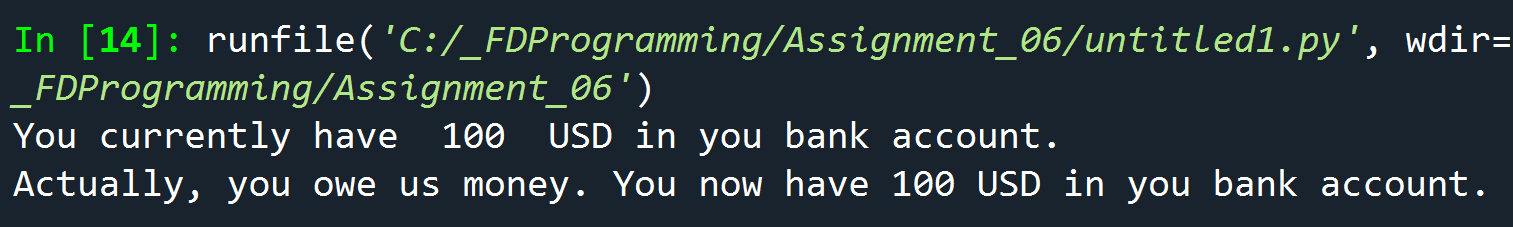


Figure - Code output of Figure 12

Best Practices

In general, when it is not a good idea to shadow variables inside functions as it can lead to confusion. Along the same vein, using global variables should be done sparingly. Using too many of them can lead to confusion as it is hard to keep track of its changing values. **Global constants,** on the other hand, can make your code less confusing. For example, you can create a global constant called interest\_rate and set it to say 0.105. Now any calling function can just read interest\_rate variable instead of hard coding in the 0.105 value. Everything this rate changes, you only have change it in the global constant declaration rather than changing it in multiple places.

Shared References

Revisiting this important concept in this module. I found this cool [example](https://www.tutorialspoint.com/python/python_functions.htm)[[3]](#footnote-3) that explains well the difference between pass by reference vs value. Figure 14 - Pass by Reference shows how changing a variable that refers to a list outside of the function changes the value inside and outside of the function. Please see Figure 15 - Code output of Figure 14 for output.

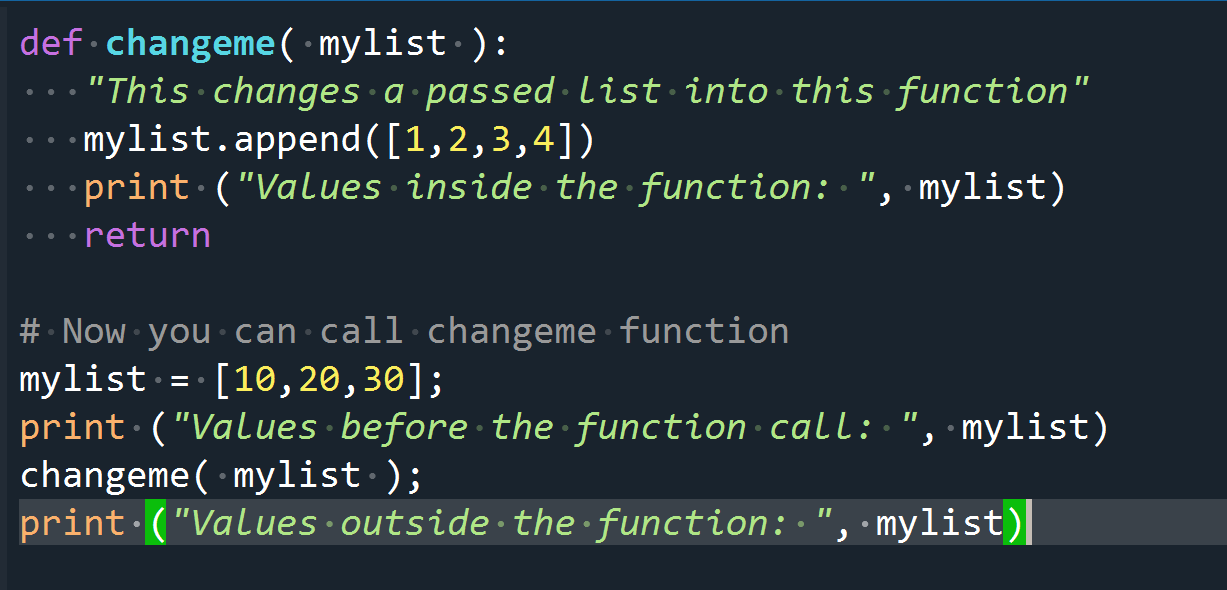


Figure - Pass by Reference

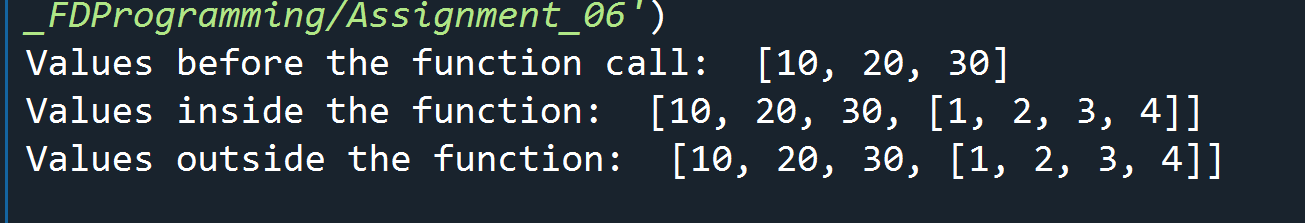


Figure - Code output of Figure 14

Now in Figure 16 - Pass by Value, mylist was getting assigned a new value inside the function which does not impact the mylist in global scope.

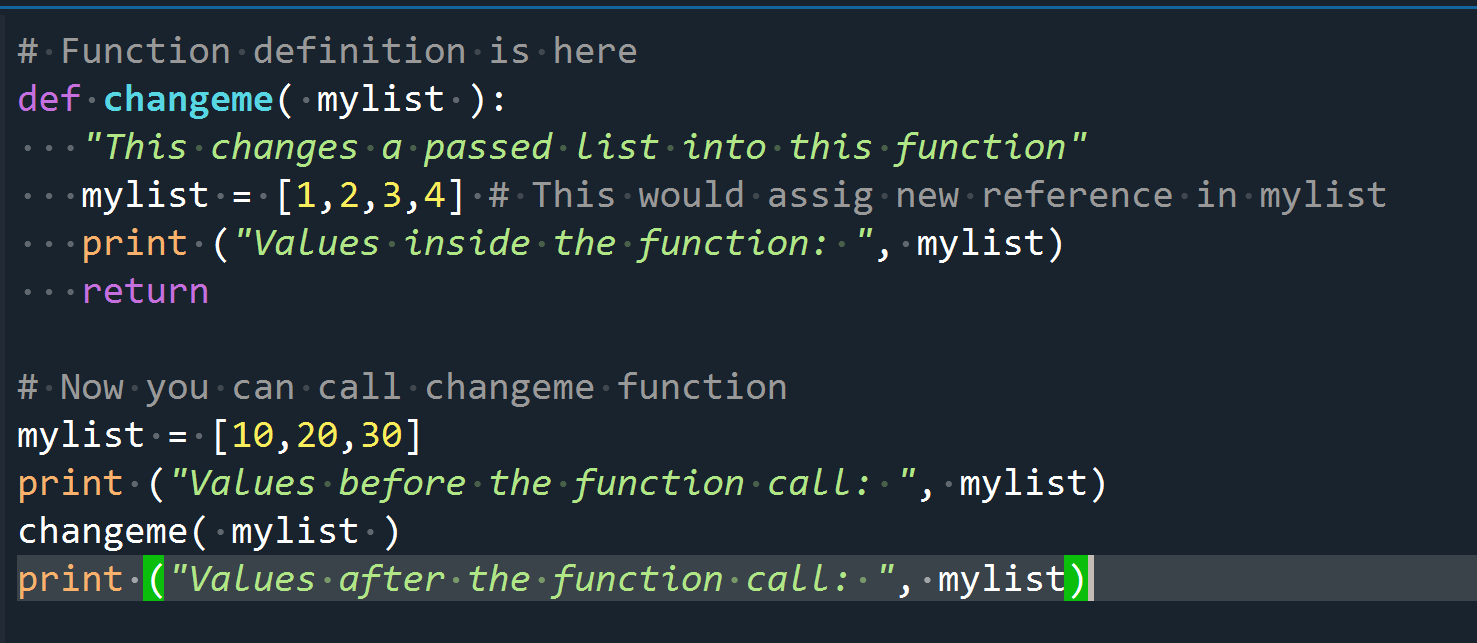


Figure - Pass by Value

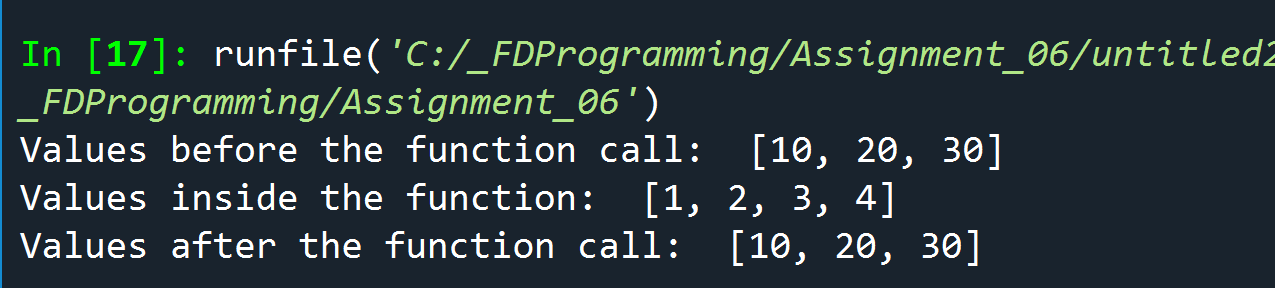


Figure - Code output of Figure 16

Classes

Classed are a way of organizing and bundling together functions based on functionality. This [webpage](https://docs.python.org/3/tutorial/classes.html) [[4]](#footnote-4) goes a little bit more in depth about classes.

For the homework, we grouped our functions into the following:

* class DataProcessor
* class FileProcessor
* class IO

As an example, in Figure 18 - class DataProcessor, I have added 2 functions called add\_newInventory and del\_inventory since these two functions take in data and makes changes to the data.

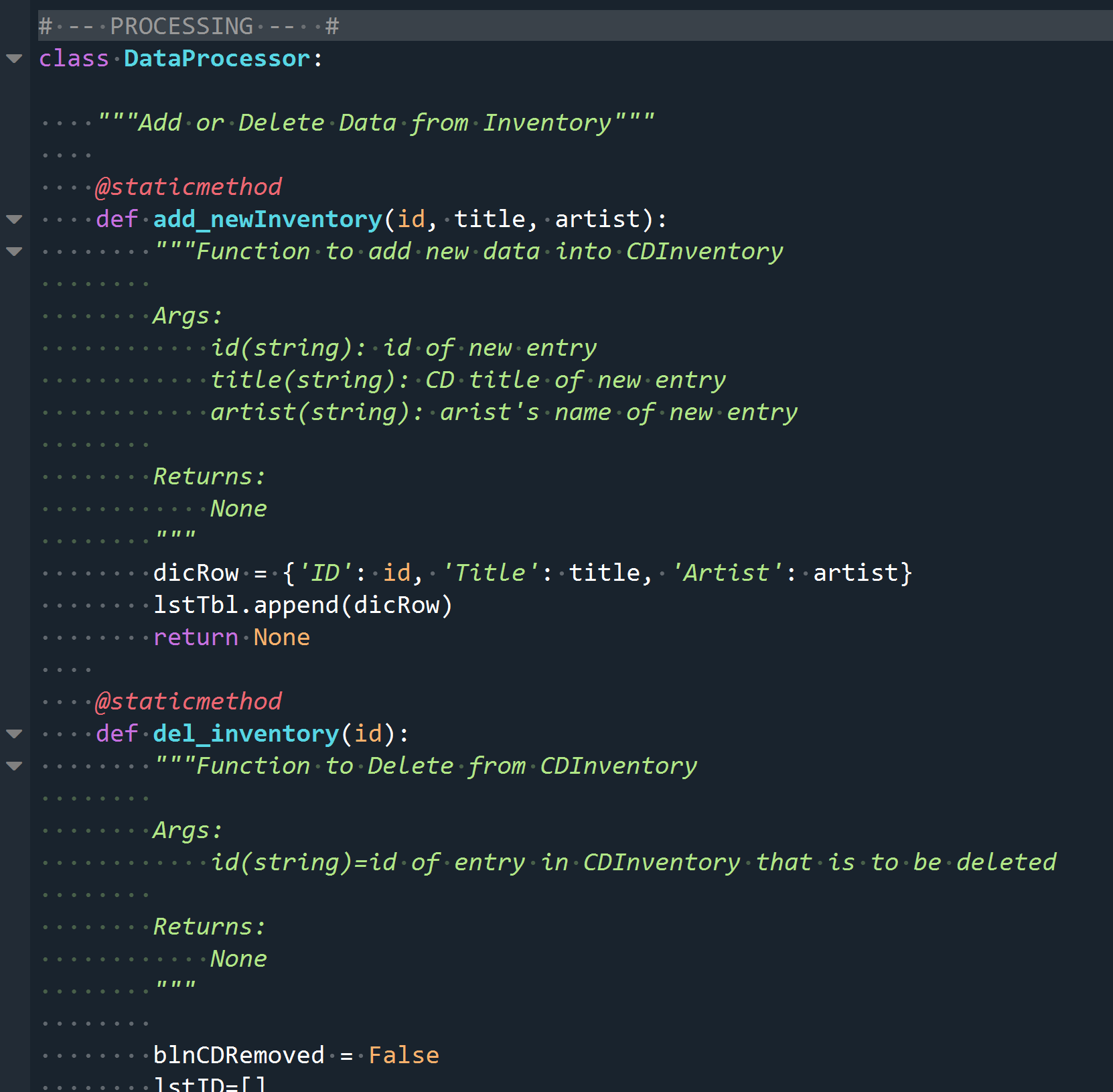


Figure - class DataProcessor

Documentation Strings

Documentation strings or **docstrings** for short, is a way of providing documentation about the purpose of a function, class, or module. It is helpful because it provides the user information about the capability and even usage of a function or module. This [website](https://www.python.org/dev/peps/pep-0257/) discusses in more detail some best practices to observe when defining one-liner (Figure 19 - One-Liner Doc Strings) or multi-liner doc strings. Referring back to Figure 18 - class DataProcessor, I employed a one-liner doc string after the class definition as well as a multi-line doctring after the definition of the function. The doc strings would be the string enclosed in triple double quotes (“””).

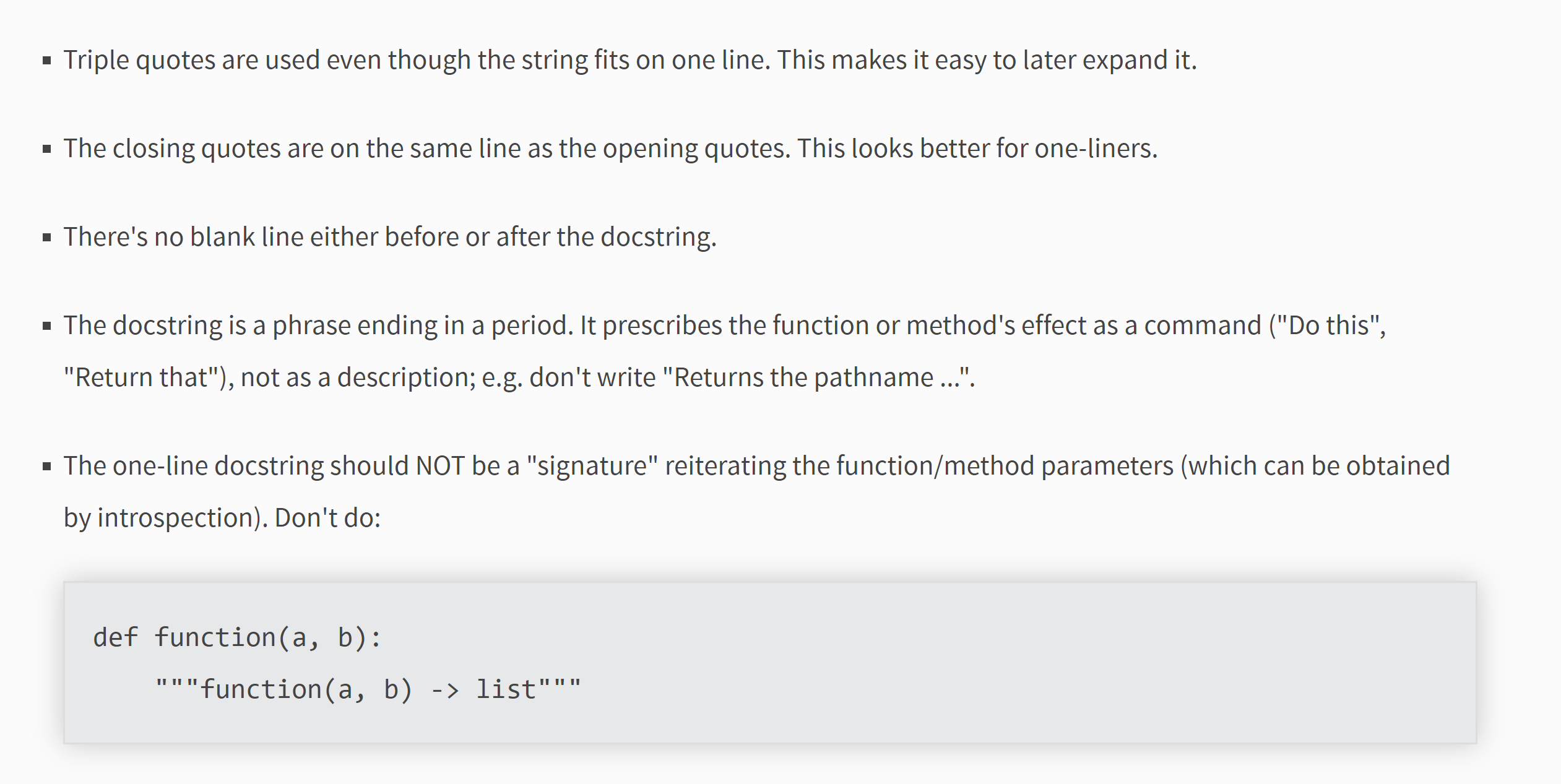


Figure - One-Liner Doc Strings

CDInventory Program

Please refer to Figure 23 - CDInventory in Spyder Console and Figure 24 - CDInventory in Windows Console to see CDInventory.py in action,

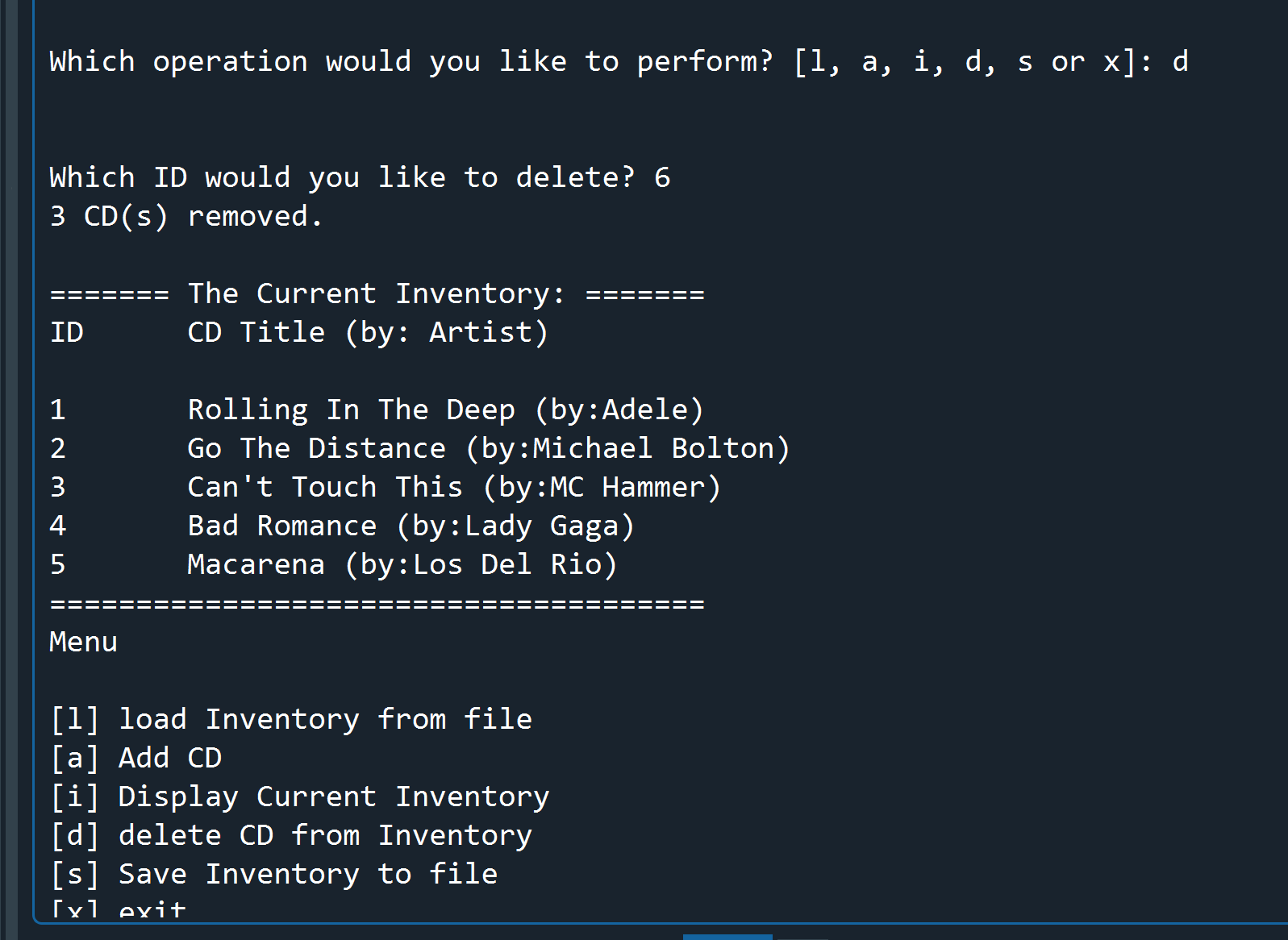


Figure - CDInventory in Spyder Console

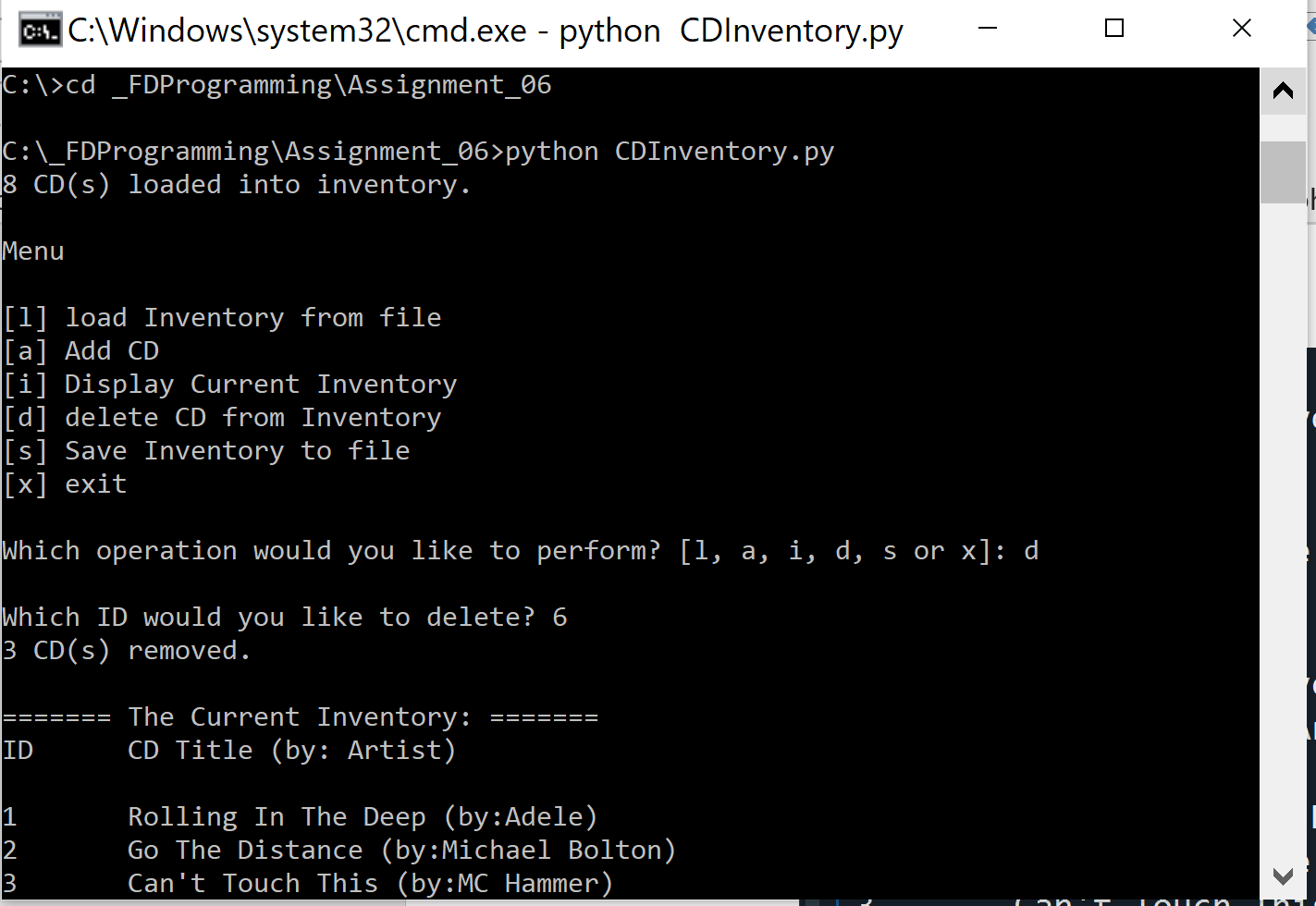


Figure - CDInventory in Windows Console

# Summary

I have learned a lot of new things in this homework. I have learned how to define and call functions. I also learned how to pass arguments to a function via parameters. I learned the when using positional arguments, the order of how you pass on the values matter whereas when using keyword arguments, it does not. I learned that variables defined in functions are only accessible by that function and can’t be seen or accessed by other functions. Global variables, on the other hand, can be accessed by functions and maybe change but only if the keyword globalis used. I gained a lot of knowledge regarding best practices such as using classes to group functions by functionality and using docstrings to succinctly describe what the class or function is for.

# Appendix

1. CDInventory.py
2. #------------------------------------------#
3. # Title: Assignment06.py
4. # Desc: Working with classes and functions.
5. # Change Log: (Who, When, What)
6. # Maria Dacutanan, 2020-Aug-16, Updated read\_file function to include error handling for file not existing
7. # Maria Dacutanan, 2020-Aug-16, Updated show\_inventory function to include Check for empty table
8. # Maria Dacutanan, 2020-Aug-16, Added get\_newInventory function in class IO
9. # Maria Dacutanan, 2020-Aug-16, Added add\_newInventory function in class DataProcessor
10. # Maria Dacutanan, 2020-Aug-16, Added code for write\_file function in class FileProcessor
11. # Maria Dacutanan, 2020-Aug-16, Added del\_inventory function in class DataProcessor
12. # Maria Dacutanan, 2020-Aug-18, Updated del\_inventory function to delete duplicate entries
13. #------------------------------------------#
15. # -- DATA -- #
16. strChoice = '' # User input
17. lstTbl = []  # list of lists to hold data
18. dicRow = {}  # list of data row
19. strFileName = 'CDInventory.txt'  # data storage file
20. objFile = None  # file object
21. loadErr=False
23. # -- PROCESSING --  #
24. **class** DataProcessor:
26. """Add or Delete Data from Inventory"""
28. @staticmethod
29. **def** add\_newInventory(id, title, artist):
30. """Function to add new data into CDInventory
32. Args:
33. id(string): id of new entry
34. title(string): CD title of new entry
35. artist(string): arist's name of new entry
37. Returns:
38. None
39. """
40. dicRow = {'ID': id, 'Title': title, 'Artist': artist}
41. lstTbl.append(dicRow)
42. **return** None
44. @staticmethod
45. **def** del\_inventory(id):
46. """Function to Delete from CDInventory
48. Args:
49. id(string)=id of entry in CDInventory that is to be deleted
51. Returns:
52. None
53. """
55. blnCDRemoved = False
56. lstID=[]
57. delctr=0
58. **for** cd **in** lstTbl:
59. **for** row **in** cd['ID']:
60. lstID.append(row) #Store all IDs from lstTbl into lstID table
61. **if** lstID.count(id) > 0: #Check if user input exists in lstID
62. intRowNr = 0
63. #This while block will loop thru lstTbl to delete ALL instances of ID in case of duplicates
64. **while** intRowNr < len(lstTbl):
65. **if** (lstTbl[intRowNr]['ID']) == id:
66. **del** lstTbl[intRowNr]
67. delctr+=1 #Count number of deletions
68. intRowNr=0 #if ID was deleted, restart intRowNr as lstTbl has shifted
69. **else**:
70. intRowNr += 1#increase intRowNr to move on to next index of lstTbl
71. blnCDRemoved = True
73. **if** blnCDRemoved:
74. **print**('{} CD(s) removed.\n'.format(delctr))
75. **else**:
76. **print**('Could not find this CD!\n')
77. **return** None

80. **class** FileProcessor:
81. """Processing the data to and from text file"""
83. @staticmethod
84. **def** read\_file(file\_name, table):
85. """Function to manage data ingestion from file to a list of dictionaries
87. Reads the data from file identified by file\_name into a 2D table
88. (list of dicts) table one line in the file represents one dictionary row in table.
90. Args:
91. file\_name (string): name of file used to read the data from
92. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
94. Returns:
95. None.
96. """
97. table.clear()  # this clears existing data and allows to load data from file
98. loadctr=0
100. **try**:
101. objFile=open(file\_name, 'r') #open CDInventory.txt and store in objfile
102. **for** line **in** objFile:
103. data = line.strip().split(',')
104. dicRow = {'ID': str(data[0]), 'Title': data[1], 'Artist': data[2]}
105. table.append(dicRow)
106. loadctr+=1 #count number of rows loaded into memory
107. **print** ('{} CD(s) loaded into inventory.\n'.format(loadctr))
108. **return** None
109. **except**:
110. **print**('Unable to load inventory from ' + file\_name + '.\n') #if unable to load file, return error msg and break out of loop
111. **return** None
112. objFile.close()
114. @staticmethod
115. **def** write\_file(file\_name, table):
116. """Function to Save CDInventory into File
118. Args:
119. file\_name(file object)=filename of CDInventory file
120. table(list)= list of CDInventory dictionaries
122. Return:
123. None
124. """
126. savectr=0
127. objFile = open(file\_name, 'w')
128. **for** row **in** table:
129. lstValues = list(row.values())
130. objFile.write(','.join(lstValues) + '\n')
131. savectr+=1 #counts number of rows saved into file
132. objFile.close()
133. **print** ('{} CD(s) saved into {}.\n'.format(savectr,file\_name))
134. **return** None

137. # -- PRESENTATION (Input/Output) -- #
139. **class** IO:
140. """Handling Input / Output"""
142. @staticmethod
143. **def** print\_menu():
144. """Displays a menu of choices to the user
146. Args:
147. None.
149. Returns:
150. None.
151. """
153. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
154. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
156. @staticmethod
157. **def** menu\_choice():
158. """Gets user input for menu selection
160. Args:
161. None.
163. Returns:
164. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
166. """
167. choice = ' '
168. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
169. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
170. **print**()  # Add extra space for layout
171. **return** choice
173. @staticmethod
174. **def** show\_inventory(table):
175. """Displays current inventory table

178. Args:
179. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
181. Returns:
182. None.
184. """
185. **if** (table):
186. **print**('======= The Current Inventory: =======')
187. **print**('ID\tCD Title (by: Artist)\n')
188. **for** row **in** table:
189. **print**('{}\t{} (by:{})'.format(\*row.values()))
190. **print**('======================================')
191. **else**:
192. **print** ('Inventory is empty.\n')
193. **return** None
195. @staticmethod
196. **def** get\_newInventory():
197. **while** True: #user is re-prompted for null ID
198. strID = str(input('Enter an ID: ').strip())
199. **if** (strID):
200. **break**
201. **while** True: #user is re-prompted for null CD Title
202. strTitle = input('Enter the CD\'s Title: ').strip()
203. **if** (strTitle):
204. **break**
205. **while** True: #user is re-prompted for null Artist's Name
206. strArtist = input('Enter the Artist\'s Name: ').strip()
207. **if** (strArtist):
208. **break**
209. **return** (strID, strTitle, strArtist)
211. # 1. When program starts, read in the currently saved Inventory
212. FileProcessor.read\_file(strFileName, lstTbl)
214. # 2. start main loop
215. **while** True:
216. # 2.1 Display Menu to user and get choice
217. IO.print\_menu()
218. strChoice = IO.menu\_choice()
220. # 3. Process menu selection
221. # 3.1 process exit first
222. **if** strChoice == 'x':
223. **break**
224. # 3.2 procless load inventory
225. **if** strChoice == 'l':
226. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
227. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
228. **if** strYesNo.lower() == 'yes':
229. **print**('reloading...')
230. FileProcessor.read\_file(strFileName, lstTbl) # function call to read CDInventory.txt
231. IO.show\_inventory(lstTbl)
232. **else**:
233. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
234. IO.show\_inventory(lstTbl)
235. **continue**  # start loop back at top.
237. # 3.3 process add a CD
238. **elif** strChoice == 'a':
239. # 3.3.1 Ask user for new ID, CD Title and Artist
240. intID,strTitle,strArtist=IO.get\_newInventory() #function call to prompt user for ID, CD Title and Artist and unpack return values
241. DataProcessor.add\_newInventory(intID, strTitle, strArtist) #function call to add data into inventory
242. IO.show\_inventory(lstTbl)
243. **continue**  # start loop back at top.
244. # 3.4 process display current inventory
245. **elif** strChoice == 'i':
246. IO.show\_inventory(lstTbl)
247. **continue**  # start loop back at top.
248. # 3.5 process delete a CD
249. **elif** strChoice == 'd':
250. # 3.5.1 get user input for which CD to delete
251. # 3.5.1.1 display Inventory to user
252. **if** (lstTbl): #check if lstTbl is not empty
253. # 3.5.1.2 ask user which ID to remove
254. **while** True:
255. intIDDel = input('Which ID would you like to delete? ').strip()
256. **if** (intIDDel): #user is re-prompted for empty ID
257. DataProcessor.del\_inventory(intIDDel) #function call to delete user provided ID
258. **break**
259. IO.show\_inventory(lstTbl)
260. **else**:
261. **print**('Nothing to delete. Inventory is empty.\n')
262. **continue**  # start loop back at top.
263. # 3.6 process save inventory to file
264. **elif** strChoice == 's':
265. # 3.6.1 Display current inventory and ask user for confirmation to save
266. **if** (lstTbl):
267. IO.show\_inventory(lstTbl)
268. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
269. # 3.6.2 Process choice
270. **if** strYesNo == 'y':
271. # 3.6.2.1 save data
272. FileProcessor.write\_file(strFileName, lstTbl) #function call to write inventory into file
273. **else**:
274. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
275. **else**:
276. **print**('Nothing to save. Inventory is empty.\n')
277. **continue**  # start loop back at top.
278. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be safe:
279. **else**:
280. **print**('General Error')

1. Page 168 of Python Programing for the Absolute Beginner, 3rd Edition textbook by Michael Dawson [↑](#footnote-ref-1)
2. Last retrieved 18-Aug-20 [↑](#footnote-ref-2)
3. Last retrieved 19-Aug-20 [↑](#footnote-ref-3)
4. Last retrieved 19-Aug-20 [↑](#footnote-ref-4)