

Functions & Classes

Introduction

This document is an introduction to function and classes, how to pass arguments to functions and how to return values from functions to the program. As part of the functions we will also explore the concept of global and local variables. For this coding assignment a starter script with established TODOs was provided. The starter script had multiple classes and functions. The TODOs in the starter script were to either create new functions or using existing statements and turning them into functions.

Part 1 – Knowledge

Part 1 of this document explains key concepts learned in this module as outlined in the introduction.

What is a function?

Functions are multiple statements grouped together that can be accessed by calling the function name. When a function gets called all the statements within the function get called and executed. Once this has been completed the runtime jumps back to the location where the function was called and continues the execution.

Creating functions helps the programmer to split up complex programs into smaller, manageable pieces and make it easier for the programmer to understand the program. Using functions also improve the separation of concerns as the functions can be defined in the processing part of the program. Functions can be written in the way they will return a value, and this end value then can be returned in the I/O section of the program, further improving the separation of concerns.

What are parameters, arguments and what is the difference between these too?

Parameters and arguments are used when defining and calling a function. When writing a function, the programmer can define and use variables within the function. In the listing below from Lab06-A the parameters are 'value1' and 'value2':

```
1. def getSum(value1, value2):  
2.     return value1 + value2
```

Listing 1 - Function Definition

After a function has been defined it can be called. When calling a function values can be passed to the function. These values are called arguments and are assigned to the parameters in the function. In the example below the user is asked for two input values that are assigned to the variables 'intNumA' and 'intNumB'. The values of these variables are then used as arguments when the function gets called in line 3 and are assigned to the parameters 'value1' and 'value2' in the defined function.

```
1. intNumA = int(input('Please enter the 1st number: '))  
2. intNumB = int(input('Please enter the 2nd number: '))  
3. print('Sum:\t\t', getSum(intNumA, intNumB),
```

Listing 2 - Function Call with Arguments

A function can be called multiple times in a program so the arguments passed to the function can change throughout the program execution. Often the words parameters and arguments are used interchangeable.

To summarize: arguments are the values that are being passed to a function when the function is called while the parameters are the variables used in the function definition.

Listing 1 and 2 showed the use of positional parameters and arguments. Listing 3 shows an improvement of Listing 1 and 2 where the positional arguments have been replaced with keyword arguments:

```
1. def getSum(value1, value2):
2.     return value1 + value2
3.
4. intNumA = int(input('Please enter the 1st number: '))
5. intNumB = int(input('Please enter the 2nd number: '))
6. print('Sum:\t\t', getSum(value1 = intNumA, value2 = intNumB))
```

Listing 3 - Keyword Arguments

Keyword Arguments allow the direct assignment of the arguments to the parameters in the function rather than relying on the position of the arguments passed. Functions also allow to assignment of default values in case the function is called without any arguments for example: `def getSum(value1 = 1, value2 = 2)`

For further information about parameters and arguments see [Wikipedia](#)¹ and [Quora](#)²

What are return values?

Return values can be part of functions as seen in Listing 1. When the function gets called and executed the programmer can use the return command to pass any resulting value back. This functionality can be seen in Listing 2 in line 3. Here the function is being called with two arguments. The function in Listing 1 then gets executed with the passed parameters and returns a value. As soon as the function executes the return command the function ends. The return value then gets printed in line 3 of Listing 2.

The description above is for a function that returns one value. Functions can also return 'None' or multiple values. A function returns 'None' when the return keyword is not used in the function or the return keyword is used without anything else after 'return'.

If a function is defined with multiple return values Python automatically packages these values in a tuple, this is an implicit Python behavior. The return value of the function then will be tuple that needs to be unpacked into variables to be accessed.

What is the difference between a global and a local variable and what is shadowing?

Before working with functions, we only worked with global variables defined in the program. Functions add an additional layer to this and introduce local variables. Local variables only available within a function and cannot be called outside a function.

Global Variable

- Global variables can be called within a function, A they are available 'globally' and not just outside a function.
- Per Python default behavior the value of global variables cannot be changed

¹ [https://en.wikipedia.org/wiki/Parameter_\(computer_programming\)](https://en.wikipedia.org/wiki/Parameter_(computer_programming)) – Retrieved 2020-Feb-29

² <https://www.quora.com/What-is-the-difference-between-parameters-and-arguments-in-Python> - Retrieved 2020-Feb-

Local Variable

- Local variables cannot be called outside the local function they are declared in and are only available within their function
- Manipulating a global variable within a function is possible with the keyword 'global' in front of the variable name.
 - However, this is discouraged and should not be used as this goes against the concept of encapsulation
 - If the keyword 'global' is not used the function will not reference the global variable and will instead create a new, local variable with the same name. This is called shadowing and is not recommended as it easily creates confusion for the programmer.

How do you use functions to organize your code?

Functions allow abstraction of the code as the details of code are encapsulated in a function. Functions allow the programmer to split and organize the program up in small, manageable pieces that can be called independently. Doing so allows the programmer then to call each of these individual functions in the main body of the code without having to read through all the details of each individual function, further improving the readability of the program.

What is the difference between a function and a class?

Classes are a way to group functions whereas functions are used to group multiple statement. The functions within a class then can be individually called by calling the class and then followed by the function name, e.g. 'class.function()'. Classes also have the possibility to be imported with an import statement from an external file.

How do functions help you program using the "Separations of Concerns" pattern?

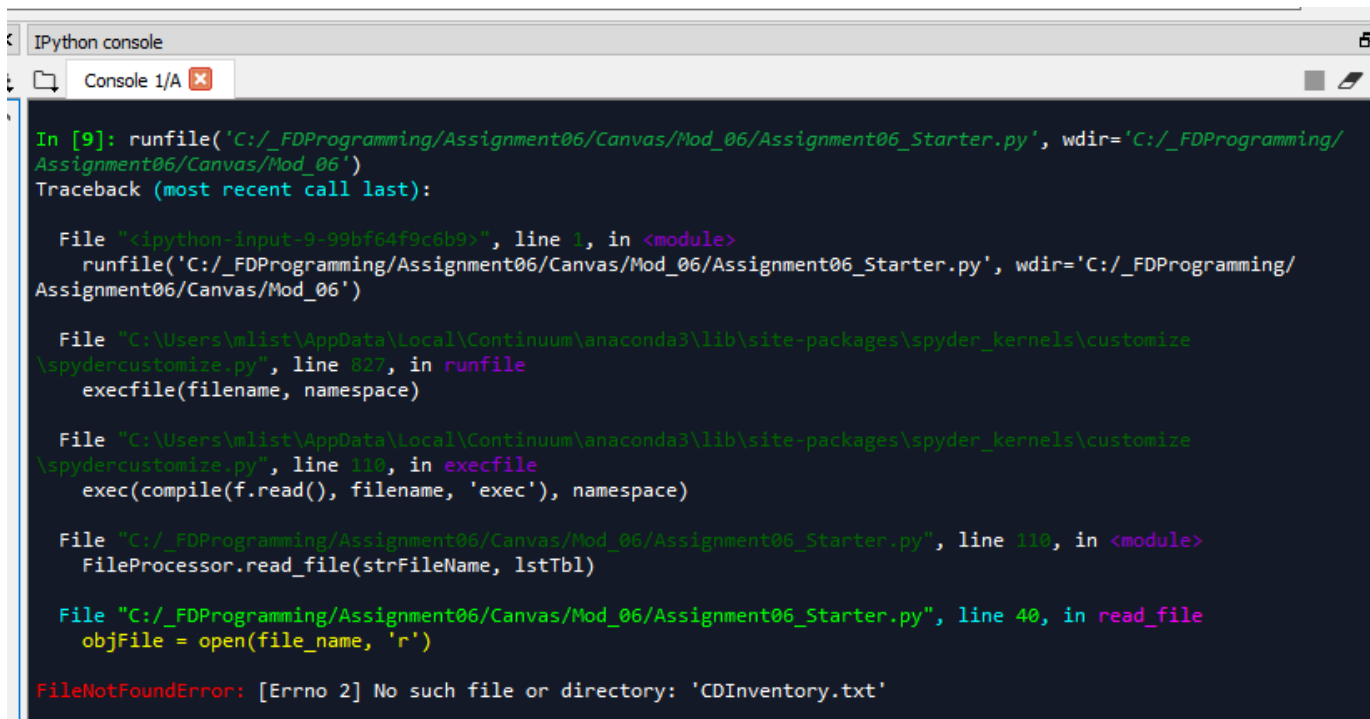
Functions can return values and be defined in the processing section of the program and then called later in the main body of the program in the I/O section. By using the return functionality of functions, the whole processing of any kind of data can be moved in the processing section and the I/O section will then just retrieve any result values from the functions. Doing so will totally separate these concerns.

Part 2 – Address Book Python Script

Part 2 of this assignment was to further modify the CD inventory program from last week's assignment. However, instead of modifying our own code the students have been provided with a starter code to gain working experience with unfamiliar code. The starter code included seven TODOs, all of them were about turning existing statements in the main body of the script into functions and calling the functions in the main body instead. My assignment solution can be found at https://github.com/List-Michael/Assignment_06³

Before I started to edit the starter script, I tried to run the starter script and it immediately crashed after starting as can be seen in the screenshot on the next site.

³ Created 2020-Mar-01



```
IPython console
Console 1/A x

In [9]: runfile('C:/_FDProgramming/Assignment06/Canvas/Mod_06/Assignment06_Starter.py', wdir='C:/_FDProgramming/Assignment06/Canvas/Mod_06')
Traceback (most recent call last):

  File "<ipython-input-9-99bf64f9c6b9>", line 1, in <module>
    runfile('C:/_FDProgramming/Assignment06/Canvas/Mod_06/Assignment06_Starter.py', wdir='C:/_FDProgramming/Assignment06/Canvas/Mod_06')

  File "C:\Users\mlist\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py", line 827, in runfile
    execfile(filename, namespace)

  File "C:\Users\mlist\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py", line 110, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/_FDProgramming/Assignment06/Canvas/Mod_06/Assignment06_Starter.py", line 116, in <module>
    FileProcessor.read_file(strFileName, lstTbl)

  File "C:/_FDProgramming/Assignment06/Canvas/Mod_06/Assignment06_Starter.py", line 40, in read_file
    objFile = open(file_name, 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'CDInventory.txt'
```

Figure 1 - Assignment06_Starter.py crashing

The starter script tries to open the CDInventory.txt file in the read_file() function and crashes if it does not exist. Adding a try-except statement as seen in the listing below solved this issue:

```
1. def read_file(file_name, table):
2.     """Function to manage data ingestion from file to a list of dictionaries
3.
4.     Reads the data from file identified by file_name into a global 2D table
5.     (list of dicts) table one line in the file represents one dictionary row in table.
6.
7.     Args:
8.         file_name (string): name of file used to read the data from
9.         table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
10.
11.     Returns:
12.         None.
13.     """
14.     table.clear() # this clears existing data and allows to load data from file
15.     try:
16.         with open(file_name, 'r') as objFile:
17.             for line in objFile:
18.                 data = line.strip().split(',')
19.                 dicRow = {'ID': int(data[0]), 'Title': data[1], 'Artist': data[2]}
20.                 table.append(dicRow)
21.     except IOError:
22.         print('\nThere is currently no existing inventory file\n')
```

Listing 4 – Improved read_file function

After resolving this issue, I attempted to understand the starter script logic before attempting to change anything. However, besides feeling confident after reviewing the material I hit a major roadblock in understanding the script.

The part of the starter script I could not follow was the function `read_file()`:

```
1. def read_file(file_name, table):
2.     """Function to manage data ingestion from file to a list of dictionaries
3.
4.     Reads the data from file identified by file_name into a 2D table
5.     (list of dicts) table one line in the file represents one dictionary row in table.
6.
7.     Args:
8.         file_name (string): name of file used to read the data from
9.         table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
10.
11.     Returns:
12.         None.
13.     """
14.     table.clear() # this clears existing data and allows to load data from file
15.     objFile = open(file_name, 'r')
16.     for line in objFile:
17.         data = line.strip().split(',')
18.         dicRow = {'ID': int(data[0]), 'Title': data[1], 'Artist': data[2]}
19.         table.append(dicRow)
20.     objFile.close()
```

Listing 5 - Original `read_file` function from the starter script

This function is not using the return statement and I did not understand why this class was able to properly read out a file and ending up storing this information in a global variable. This made me question my overall understanding of functions, so I went back to review the material to fully understand what is going on here. It took me a while to realize that the variable 'table' within the function is referencing the same memory address as the global variable 'lstTbl' that is passed as an argument in the function call.

By trying to understand this behavior in the starter script I believe I was fully able to wrap my head around on how functions and variable scope works, especially together with global variables of the reference datatype. With this knowledge I had no difficulty completing the TODOs in the starter script and applied the learned knowledge by updating the global 2D list without relying on return values when applicable.

I think my initial confusion also stemmed from the book material which I reviewed after the videos and the PDF. In the book ⁴ on page 183 it says "Changing a mutable parameter directly like this is considered creating a *side effect*. Not all side effects are bad, but this type is generally frowned upon (I'm frowning right now, just thinking about it). It's best to communicate with the rest of your program through **return values**; that way, it's clear exactly what information you're giving back."

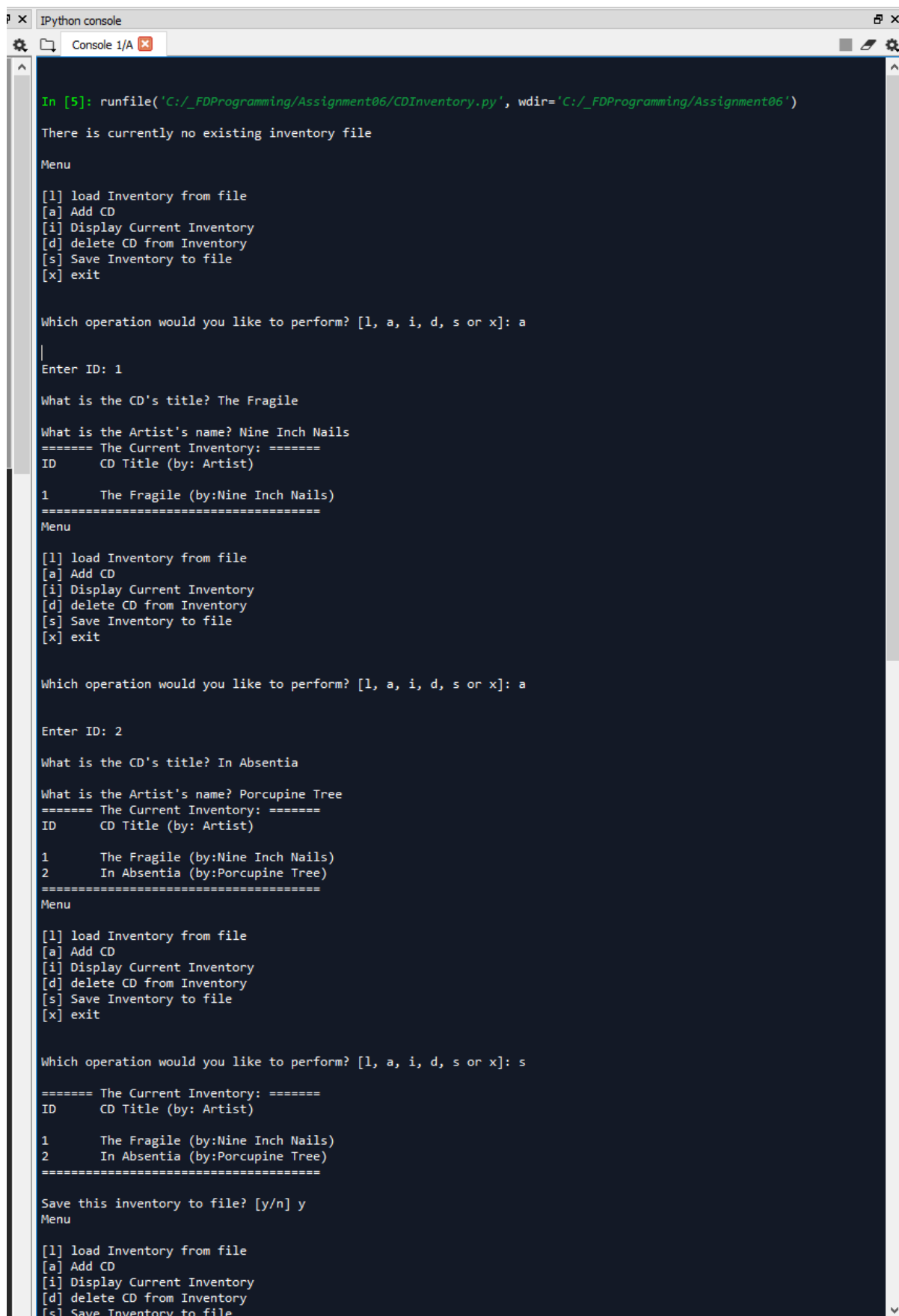
This statement describes the issue I experienced, because when I was initially reviewing the function it was not clear to me what the function was returning or manipulating. The implementation of return values rather than manipulating the reference-type list would have been clearer in my opinion. On Page 14 of our pdf ⁵ it also says, "Especially as doing this unintended **can cause a lot of grief trying to debug**". I realized that by just reading this code as a beginner.

I do understand the point Dirk made in the video series for manipulating reference-type data in functions when working with large data sets, however, it was rather confusing for a beginner like me. Overall, I think I would prefer to work with return values in functions rather than working with reference-types and would only consider manipulating reference types for larger datasets. However fully understanding this issue helped me to understand the impact reference-types do have and how powerful they can be at the same time.

⁴ Python Programming for the Absolute Beginner, Third Edition

⁵ FDN_Py_Module_06.pdf

The screenshots below show the completed CDInventory.py script running in Spyder and command line.



```
In [5]: runfile('C:/_FDProgramming/Assignment06/CDInventory.py', wdir='C:/_FDProgramming/Assignment06')

There is currently no existing inventory file

Menu

[1] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [1, a, i, d, s or x]: a

Enter ID: 1

What is the CD's title? The Fragile

What is the Artist's name? Nine Inch Nails
===== The Current Inventory: =====
ID      CD Title (by: Artist)

1       The Fragile (by:Nine Inch Nails)
=====
Menu

[1] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [1, a, i, d, s or x]: a

Enter ID: 2

What is the CD's title? In Absentia

What is the Artist's name? Porcupine Tree
===== The Current Inventory: =====
ID      CD Title (by: Artist)

1       The Fragile (by:Nine Inch Nails)
2       In Absentia (by:Porcupine Tree)
=====
Menu

[1] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [1, a, i, d, s or x]: s

===== The Current Inventory: =====
ID      CD Title (by: Artist)

1       The Fragile (by:Nine Inch Nails)
2       In Absentia (by:Porcupine Tree)
=====

Save this inventory to file? [y/n] y
Menu

[1] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
```

Figure 2 - Final CDInventory.py executed in Spyder

```
Anaconda Prompt (anaconda3)
(base) C:\_FDProgramming\Assignment06>python CDInventory.py
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [l, a, i, d, s or x]: i

===== The Current Inventory: =====
ID      CD Title (by: Artist)

1       The Fragile (by:Nine Inch Nails)
2       In Absentia (by:Porcupine Tree)
=====
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [l, a, i, d, s or x]: d

===== The Current Inventory: =====
ID      CD Title (by: Artist)

1       The Fragile (by:Nine Inch Nails)
2       In Absentia (by:Porcupine Tree)
=====
Which ID would you like to delete? 1
The CD was removed
===== The Current Inventory: =====
ID      CD Title (by: Artist)

2       In Absentia (by:Porcupine Tree)
=====
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [l, a, i, d, s or x]: s

===== The Current Inventory: =====
ID      CD Title (by: Artist)

2       In Absentia (by:Porcupine Tree)
=====
Save this inventory to file? [y/n] y
Menu

[l] load Inventory from file
[a] Add CD
[i] Display Current Inventory
[d] delete CD from Inventory
[s] Save Inventory to file
[x] exit

Which operation would you like to perform? [l, a, i, d, s or x]: x

(base) C:\_FDProgramming\Assignment06>_
```

Figure 3 - Final CDInventory.py executed in Anaconda Prompt

The below screenshot shows the created text file where the first entry has been removed as instructed in Figure 3. This shows that the file is saving correctly as well.

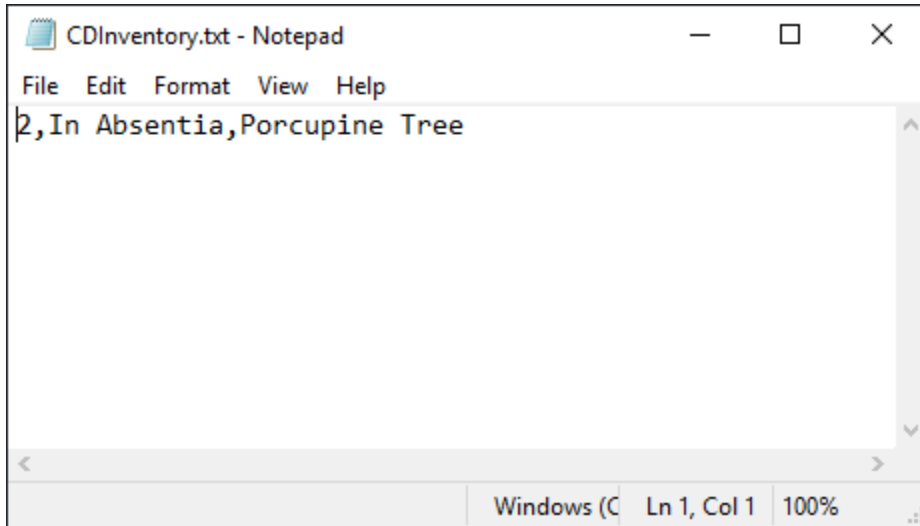


Figure 4 - CDInventory.txt after deleting

Appendix

```
1. #-----#
2. # Title: CDInventory.py
3. # Desc: Working with classes and functions.
4. # Change Log: (Who, When, What)
5. # DBiesinger, 2030-Jan-01, Created File
6. # MList, 2020-Mar-
   01, Fixed error in function read_file() as script crashes when no CDInventory.txt is present
7. # MList, 2020-Mar-
   01, Completed TODOs when 'a' is called and created functions append_cd_inventory_memory_list() + IO.c
   d_user_input()
8. # MList, 2020-Mar-
   01, Completed TODO for delete function, created delete_CD() and tested functionality
9. # MList, 2020-Mar-01, Completed TODO for saving function, build out shell function for write_file()
10. # MList, 2020-Mar-01, Ensured proper commenting in main body of script
11. # MList, 2020-Mar-01, Added docstrings for newly written functions
12. #-----#
13.
14. # -- DATA -- #
15. strChoice = '' # User input
16. lstTbl = [] # list of lists to hold data
17. dicRow = {} # dictionary of data row
18. strFileName = 'CDInventory.txt' # data storage file
19. objFile = None # file object
20.
21. #defining global variables for ID, Title and Artist to pass information between functions without sha
   dowing variables
22. strIDGlobal = ''
23. strTitleGlobal = ''
24. strArtistGlobal = ''
25.
26.
27. # -- PROCESSING -- #
28. class DataProcessor:
29.
30.     @staticmethod
31.     #result is assigned to local variable table which is referencing the same variable as the global
   lstTbl
32.     def append_cd_inventory_memory_list(strID, strTitle, strArtist, table):
33.         """Function to append new user entry as a dictionary within a 2D list
34.
35.         Uses passed arguments of CD information to create a new CD entry as a dictionary and adds di
   ctionary as a new line to global 2D table
36.
37.         Args:
38.             strID (string): Requested ID of CD the user would like to add
39.             strTitle (string): Requested CD Title the user would like to add
40.             strArtist (string): Requested Artist of CD the user would like to add
41.             table (list of dict): 2D data structure (list of dicts) that holds the data during runtim
   e
42.
43.         Returns:
44.             None.
45.         """
46.         intID = int(strID)
47.         dicRow = {'ID': intID, 'Title': strTitle, 'Artist': strArtist}
48.         table.append(dicRow)
49.
50.
51.     @staticmethod
52.     def delete_CD(ID, table):
53.         """Function to delete CD from 2D list based on ID match
54.
55.         User defines the ID of a CD the user would like to remove. ID gets passed to this function is
   used to identify a match in global 2D table.
```

```

56.         If match is identified by matching value of ID key, entry dictionary will be removed from global 2D table.
57.
58.         Args:
59.             ID (integer): Requested ID of CD the user would like to delete. Casted into integer prior
function call
60.             table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
e
61.
62.         Returns:
63.             None.
64.         """
65.         intRowNr = -1
66.         blnCDRemoved = False
67.         for row in table:
68.             intRowNr += 1
69.             if row['ID'] == ID:
70.                 del table[intRowNr]
71.                 blnCDRemoved = True
72.                 break
73.         if blnCDRemoved:
74.             print('The CD was removed')
75.         else:
76.             print('Could not find this CD!')
77.
78. class FileProcessor:
79.     """Processing the data to and from text file"""
80.     @staticmethod
81.     def read_file(file_name, table):
82.         """Function to manage data ingestion from file to a list of dictionaries
83.
84.         Reads the data from file identified by file_name into a global 2D table
85.         (list of dicts) table one line in the file represents one dictionary row in table.
86.
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) that holds the data during runtime
e
90.
91.         Returns:
92.             None.
93.         """
94.         table.clear() # this clears existing data and allows to load data from file
95.         try:
96.             with open(file_name, 'r') as objFile:
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.         except IOError:
102.             print('\nThere is currently no existing inventory file\n')
103.
104.
105.     @staticmethod
106.     def write_file(file_name, table):
107.         """Function to write added data to file
108.
109.         Reads the data of dictionaries stored in 2D table.
110.         Casts each dictionary into a string, adds a new line at the end of each string and overwrites any data in file.
111.
112.         Args:
113.             file_name (string): name of file used to read the data to
114.             table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
115.
116.         Returns:
117.             None.

```

```

118.         """
119.         objFile = open(file_name, 'w')
120.         for row in table:
121.             lstValues = list(row.values())
122.             lstValues[0] = str(lstValues[0])
123.             objFile.write(','.join(lstValues) + '\n')
124.         objFile.close()
125.
126.
127.     # -- PRESENTATION (Input/Output) -- #
128.
129.     class IO:
130.         """Handling Input / Output"""
131.
132.         @staticmethod
133.         def print_menu():
134.             """Displays a menu of choices to the user
135.
136.             Args:
137.                 None.
138.
139.             Returns:
140.                 None.
141.             """
142.
143.             print('Menu\n\n[1] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory
144.             print('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
145.
146.         @staticmethod
147.         def menu_choice():
148.             """Gets user input for menu selection
149.
150.             Args:
151.                 None.
152.
153.             Returns:
154.                 choice (string): a lower case sting of the users input out of the choices l, a, i,
155.                 d, s or x
156.
157.             """
158.             choice = ' '
159.             while choice not in ['l', 'a', 'i', 'd', 's', 'x']:
160.                 choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: '
161.                 ).lower().strip()
162.             print() # Add extra space for layout
163.             return choice
164.
165.         @staticmethod
166.         def show_inventory(table):
167.             """Displays current inventory table
168.
169.             Args:
170.                 table (list of dict): 2D data structure (list of dicts) that holds the data during
171.                 runtime.
172.
173.             Returns:
174.                 None.
175.             """
176.             print('==== The Current Inventory: =====')
177.             print('ID\tCD Title (by: Artist)\n')
178.             for row in table:
179.                 print('{ }\t{ } (by: { })'.format(*row.values()))
180.             print('=====')
181.
182.         @staticmethod

```

```

182.         def cd_user_input():
183.             """Gets user input to add new CD
184.
185.             Args:
186.                 None.
187.
188.             Returns:
189.                 strID, strTitle, strArtist (tuple): Returns tuple with the 3 variables: ID, Title
and Artist
190.
191.             """
192.             strID = input('Enter ID: ').strip()
193.             strTitle = input('What is the CD\'s title? ').strip()
194.             strArtist = input('What is the Artist\'s name? ').strip()
195.             return strID, strTitle, strArtist
196.
197.
198.         # 1. When program starts, read in the currently saved Inventory
199.         FileProcessor.read_file(strFileName, lstTbl)
200.         # 2. start main loop
201.         while True:
202.             # 2.1 Display Menu to user and get choice
203.             IO.print_menu()
204.             strChoice = IO.menu_choice()
205.             # 3. Process menu selection
206.             # 3.1 process exit first
207.             if strChoice == 'x':
208.                 break
209.             # 3.2 process load inventory
210.             if strChoice == 'l':
211.                 print('WARNING: If you continue, all unsaved data will be lost and the Inventory re-
loaded from file.')
212.                 strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will
be canceled: ')
213.                 if strYesNo.lower() == 'yes':
214.                     print('reloading...')
215.                     FileProcessor.read_file(strFileName, lstTbl)
216.                     IO.show_inventory(lstTbl)
217.                 else:
218.                     input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the
menu.')
219.                     IO.show_inventory(lstTbl)
220.                     continue # start loop back at top.
221.             # 3.3 process add a CD
222.             elif strChoice == 'a':
223.                 # 3.3.1 Ask user for new ID, CD Title and Artist and unpacking return tuple
224.                 strIDGlobal, strTitleGlobal, strArtistGlobal = IO.cd_user_input()
225.                 # 3.3.2 Add item to the table
226.                 DataProcessor.append_cd_inventory_memory_list(strIDGlobal, strTitleGlobal, strArtistGl
obal, lstTbl)
227.                 # 3.3.3 Displaying current inventory
228.                 IO.show_inventory(lstTbl)
229.                 continue # start loop back at top.
230.             # 3.4 process display current inventory
231.             elif strChoice == 'i':
232.                 IO.show_inventory(lstTbl)
233.                 continue # start loop back at top.
234.             # 3.5 process delete a CD
235.             elif strChoice == 'd':
236.                 # 3.5.1 get Userinput for which CD to delete
237.                 # 3.5.1.1 display Inventory to user
238.                 IO.show_inventory(lstTbl)
239.                 # 3.5.1.2 ask user which ID to remove
240.                 intIDDel = int(input('Which ID would you like to delete? ').strip())
241.                 # 3.5.2 search thru table and delete CD
242.                 # 3.5.2.1 delete entry
243.                 DataProcessor.delete_CD(intIDDel, lstTbl)
244.                 # 3.5.2.2 display Inventory to user

```

```

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.             FileProcessor.write_file(strFileName, lstTbl)
256.         else:
257.             input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')

258.         continue # start loop back at top.
259.     # 3.7 catch-
    all should not be possible, as user choice gets vetted in IO, but to be save:
260.     else:
261.         print('General Error')

```

Listing 6 - CDInventory.py