Maria Dacutanan

Aug 26, 2020

Foundations of Programming: Python

Assignment 07

Serialization through Pickling and Custom Error Handling

# Introduction

In module 7, I learned various way reading data from text files like reading a text file as a whole, or reading the file n characters at a time or perhaps reading it one line a time. Next, I was introduced to a brand new concept called ‘pickling’ (and ‘unpickling’) which is one of the ways that python can serialize (and deserialize) complex data which can be stored into binary files.

Next, I dig deeper into the world of structured error file handling. I gained knowledge on how to catch specific exceptions based on class as well as raising custom exceptions.

Finally, I get a peek of what markdown language is and how it is utilized in GitHub.

# Reading Data from Files

Reading N Characters from File

Most basic way of reading data from file is through the **read()** method. If read() method is used with no arguments, the entire contents of the file is read.

Observe the code shown in Figure 1 - Read Entire File. In line 22, I used the **with statement** to open and read the contents of the file vid\_games.txt (Figure 4 - vid\_games.txt) in read mode (‘r’). With the with statement, you get better syntax and exceptions handling. In addition, notice that there is no close() method used to close out the file. The with statement automatically does it for us. This [website](https://thispointer.com/python-open-a-file-using-open-with-statement-benefits-explained-with-examples/)[[1]](#footnote-1) shows examples of using the with statement as well as benefits of using it. Then in Line 23, I invoked the read() method with no arguments and stored into the data variable . In Figure 2 - Figure 1 Code Output, you see that all the contents of the file were stored as one element of the lstData list.

Now if I modify Line 23 to say ‘data.fileObj.read(3)’, the lstData now shows that I my element data is ‘Pac’.

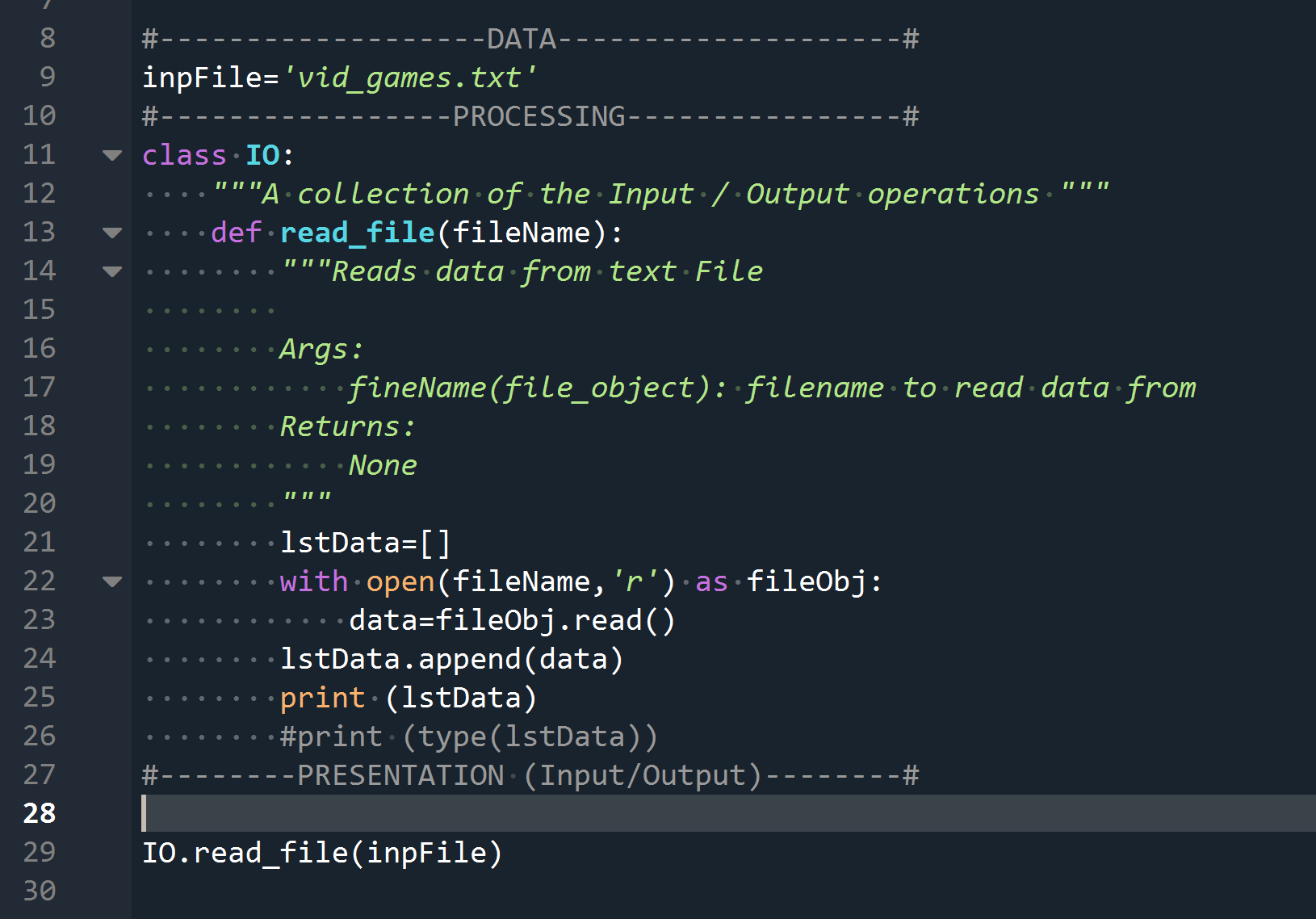


Figure 1 - Read Entire File

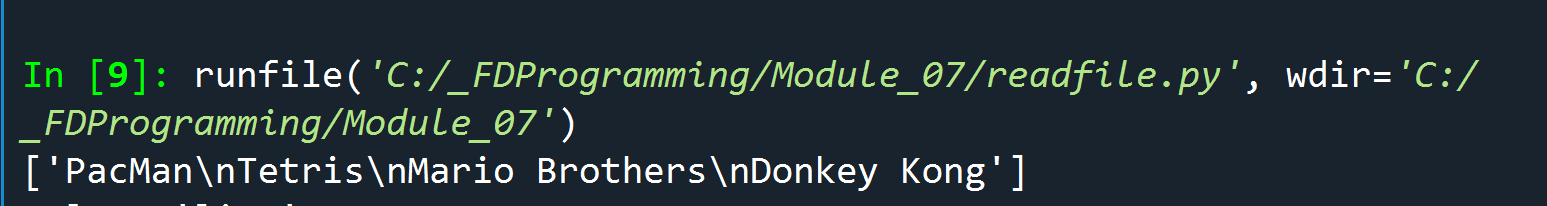


Figure 2 - Figure 1 Code Output

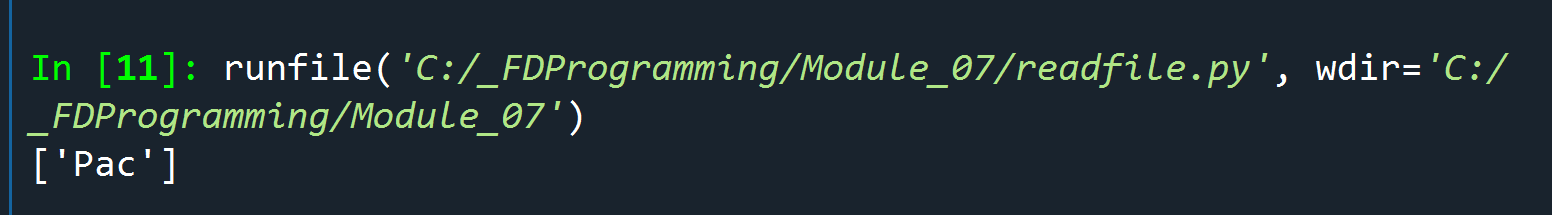


Figure 3 - Read n chars Output

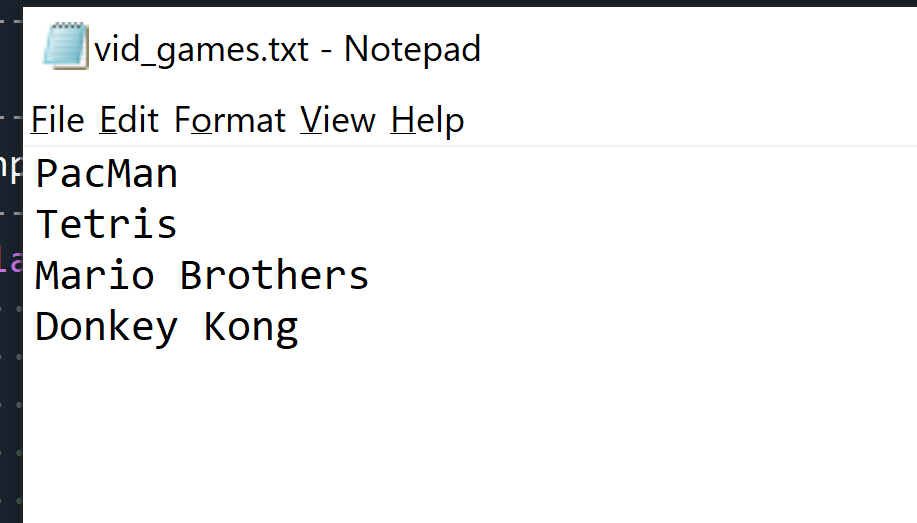


Figure 4 - vid\_games.txt

Reading N Chars Per Line of File

Now in the instance I want to read n characters in a per line basis, we can use the **readline()** method. Similar to the read() method, if the readline() method is invoked with no arguments as in Line 24 in Figure 5 - Using readline() , then the entire line is read into the line variable. As shown in Figure 6 - Output of Code in Figure 5, you can see that an element is created inside the lstData list for each line of the text file.

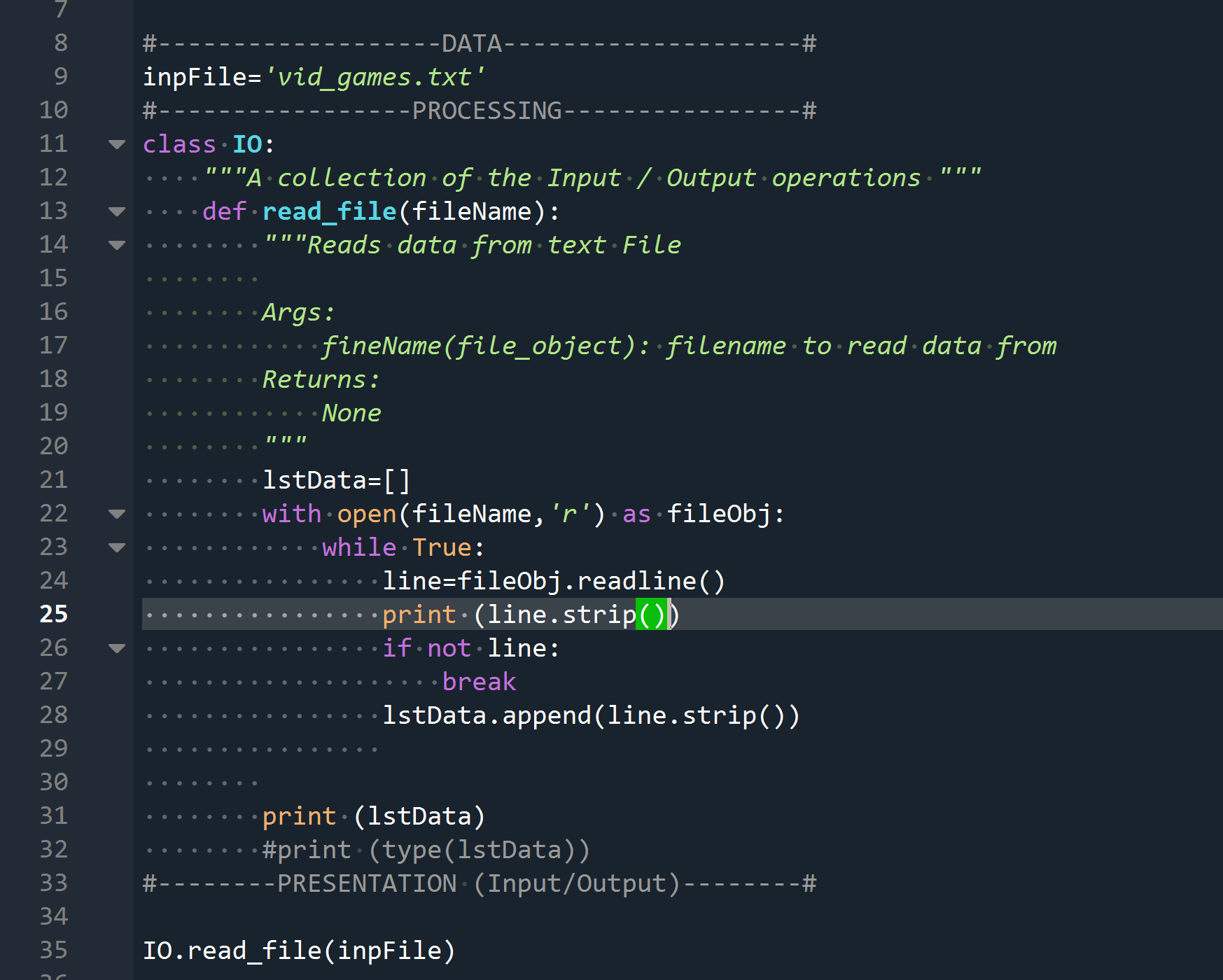


Figure 5 - Using readline() method

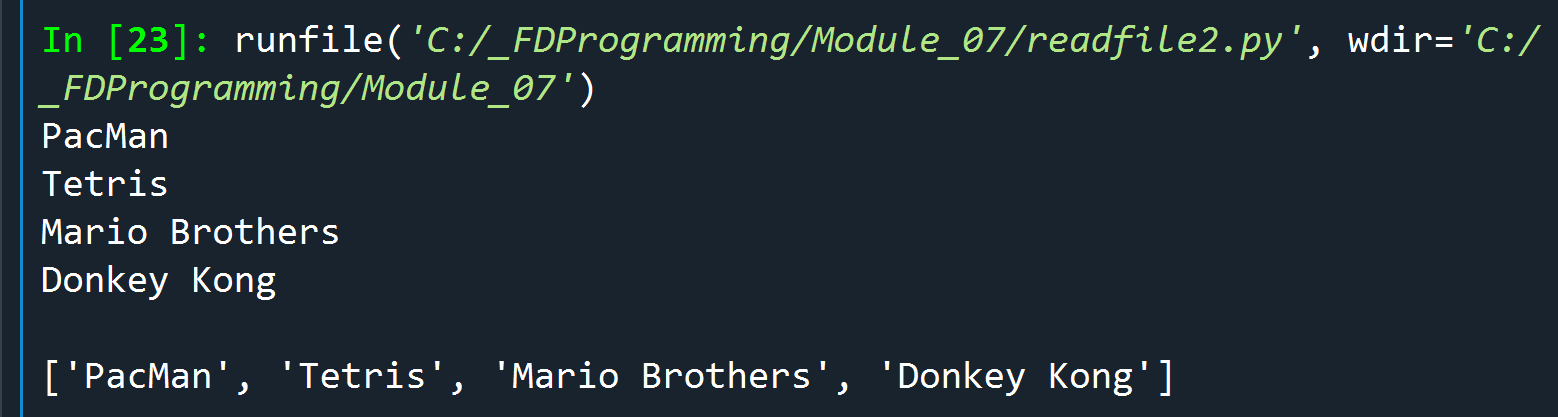


Figure 6 - Output of Code in Figure 5

Now if I modify Line 24 in Figure 5 - Using readline() so that I am reading 3 chars at a time per line (line.fileObject.readline(3)), the output of the program would like Figure 7 - Using readline() method with argument. Notice that readline(3) does not mean read the first 3 characterss of each line; rather read 3 characters at a time per line.

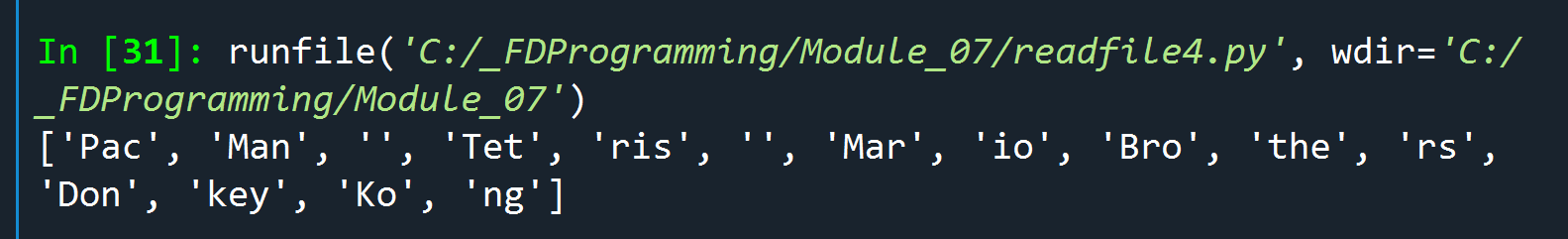


Figure 7 - Using readline() method with argument

Reading All Lines of Data into a List

Another way of reading each line of a file is thru **readfiles()** method. With this method, each line of data becomes a string element in a list. In Figure 8 - Using readlines() , line 23 shows how readlines() method is invoked and stored into the list lstData. The output of the program is shown in Figure 9 - Output of Code in Figure 8. Notice that list elements still has the newline character (\n) appended to the actual string.

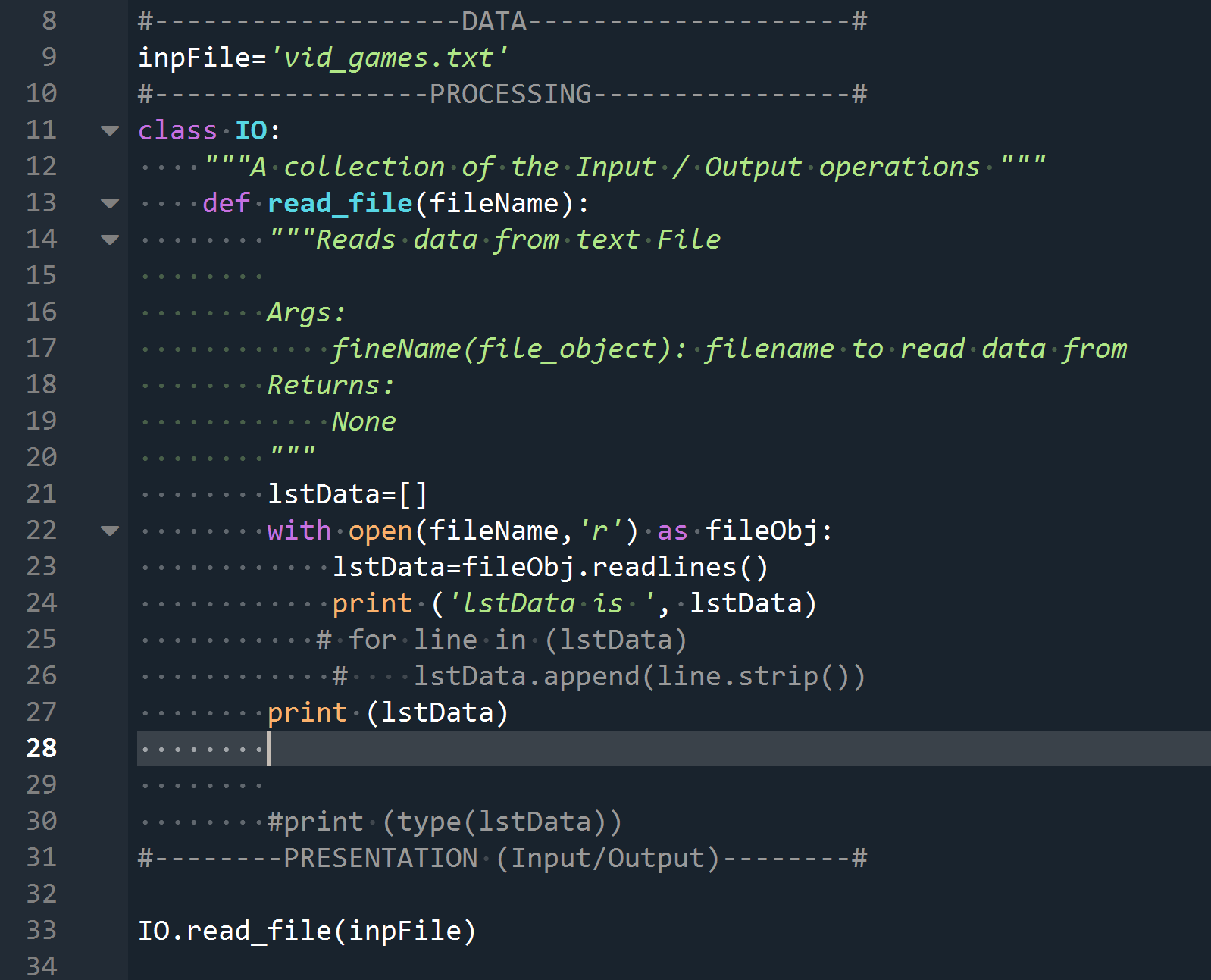


Figure 8 - Using readlines() method

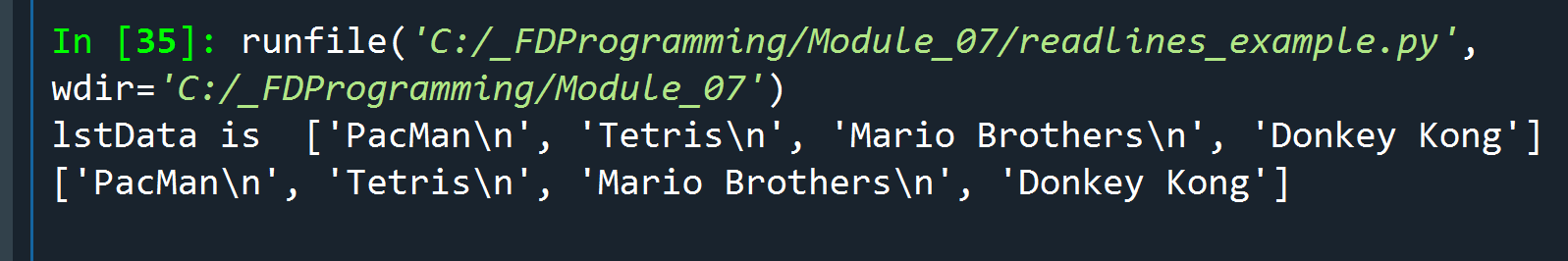


Figure 9 - Output of Code in Figure 8

If the newline characters (\n) needs to be stripped, please check out the code in Figure 10 – Using readlines() method with a for Loop. As you can see in Line 24, still using the with statement to open the file, I loop through each line of the fileObj.readlines() and then strip the newline character (\n).



Figure 10 – Using readlines() method with a for Loop

# Writing Data into Files

Writing a List of Strings into a File

In the previous module, I have already dabbled in writing data into a plain text files using the **write() method** so I won’t discuss that one in this document. However, there is another file object method called **writelines()** that I do want to talk about here. Similar to its counterpart readlines(), writelines() method writes a list of strings into a file. Before writing any daya, in Figure 11 - using writelines() method, I did some preprocessing on the data by adding the newline character (‘\n’) to each element of the list data and then I append each one to a the list lstData. Finally, in line 31, I invoke the writelines() method to write lstData into file.

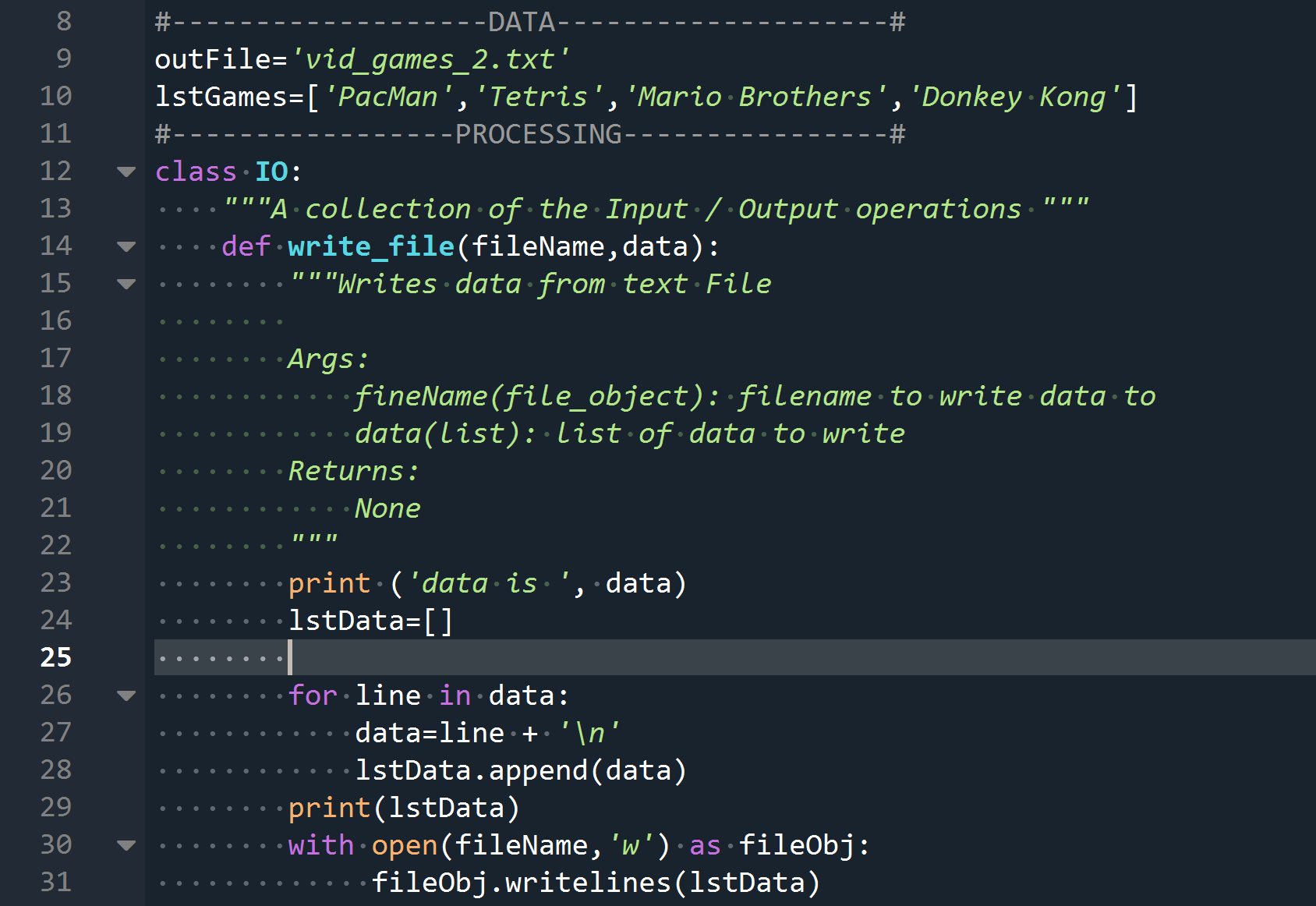


Figure 11 - using writelines() method

# Serialization and Deserialization of Data

In computing, **serialization** (US spelling) or serialisation (UK spelling) is the process of translating a [data structure](https://en.wikipedia.org/wiki/Data_structure) or [object](https://en.wikipedia.org/wiki/Object_(computer_science)) state into a format that can be stored (for example, in a [file](https://en.wikipedia.org/wiki/Computer_file) or memory [data buffer](https://en.wikipedia.org/wiki/Data_buffer)) or transmitted (for example, across a [computer network](https://en.wikipedia.org/wiki/Computer_network)) and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.

This process of serializing an object is also called [marshalling](https://en.wikipedia.org/wiki/Marshalling_(computer_science)) an object in some situations. The opposite operation, extracting a data structure from a series of bytes, is deserialization, (also called unserialization or [unmarshalling](https://en.wikipedia.org/wiki/Unmarshalling)).[[2]](#footnote-2)

# Serialization and Deserialization in Python

So now that we have the official Wikipedia definition of what serialization and deserialization out the way, how does Python implement this process? Python has a couple of ways of serialization in in Python and for this homework, we will delve in the world of pickling.

Pickling in Python

When I hear of pickling, I think about pickled vegetables in jars. And the reason people pickle vegetables back in the day is to preserve food and extend its shelf life for later use. Well, pickling in Python is the same idea; preserving python objects for later use.

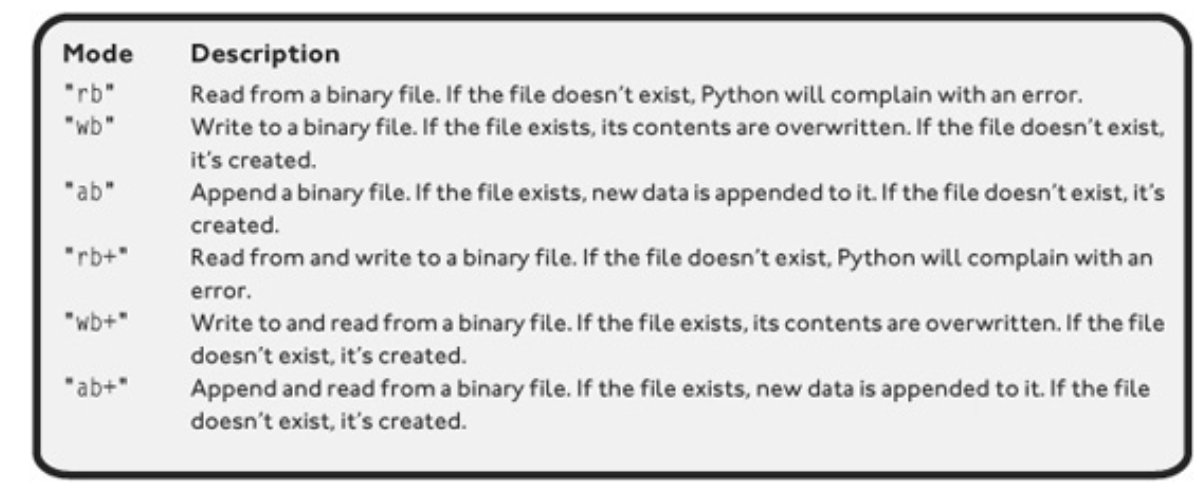
**Pickling** is the process whereby a Python object hierarch is converted into a byte stream (usually no human readable) to be written into a file **Unpickling** is the reverse operation whereby a byte stream is converted back into a working Python object hierarchy.[[3]](#footnote-3)

Pickling Data and Writing Into A File

So how do we pickle objects I Python? First, you need to import the pickle module

**import pickle**

Next we need to open a binary file where we intend to store data. Here are the different binary file access methods:



Next, we use the **dump()** function to pickle data.

*Pickle module has four available methods:*

* ***dump()****− Serializes to an open file (file-like object).*
* ***dumps()****− Serializes to a string.*
* ***load()****− Deserializes from an open-like object.*
* ***loads()****− Deserializes from a string.*

Figure 12 - using dump() method to pickle data illustrates a function I created for Lab07\_C. The function is used to write the results of math calculations into a .dat file.

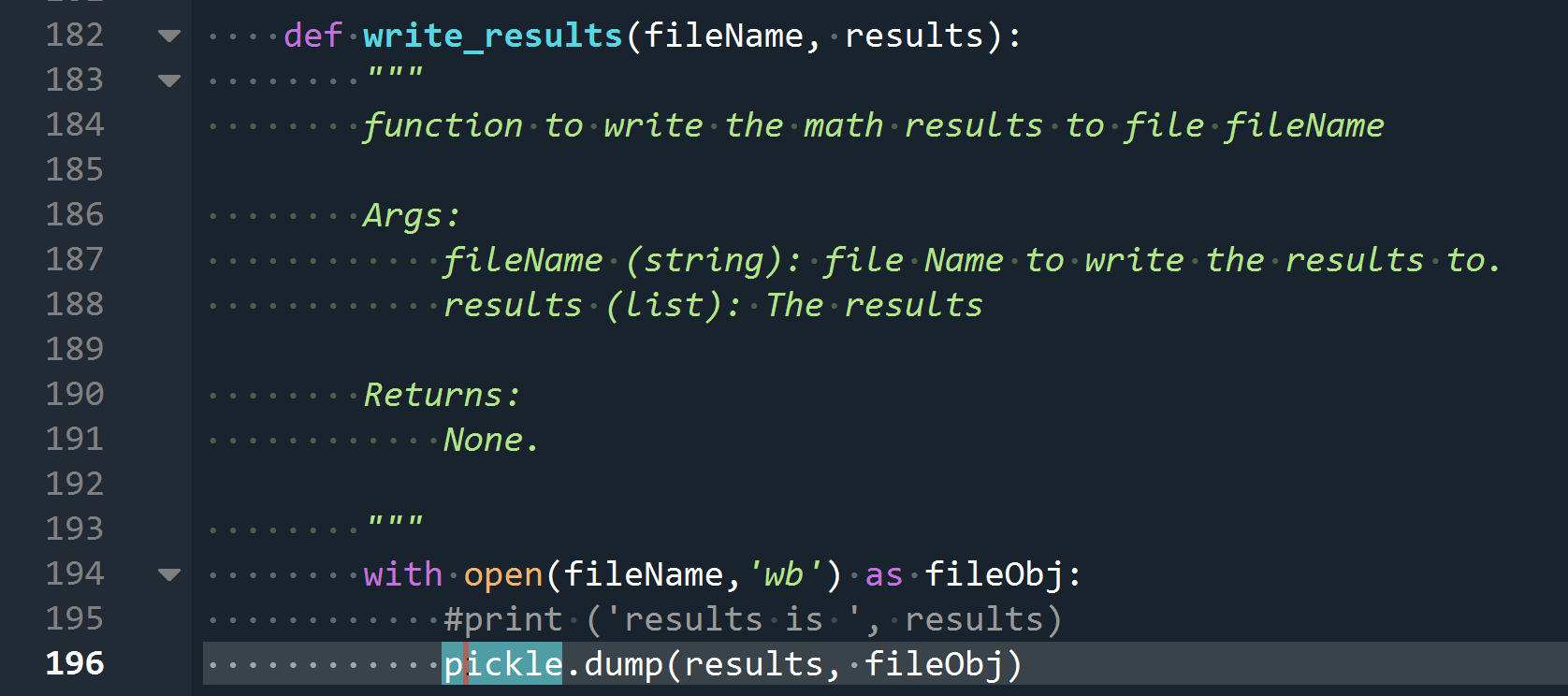


Figure 12 - using dump() method to pickle data

Unpickling Data from a Binary File

Unpickling process is similar to pickling. Figure 13 - Unpickling data shows a function I created for Lab07\_C. As you can see, first I have to open the binary file and then use the **load() method.**

If you are curious what a pickled file looks like, check out Figure 14 - Pickled data (binary file)

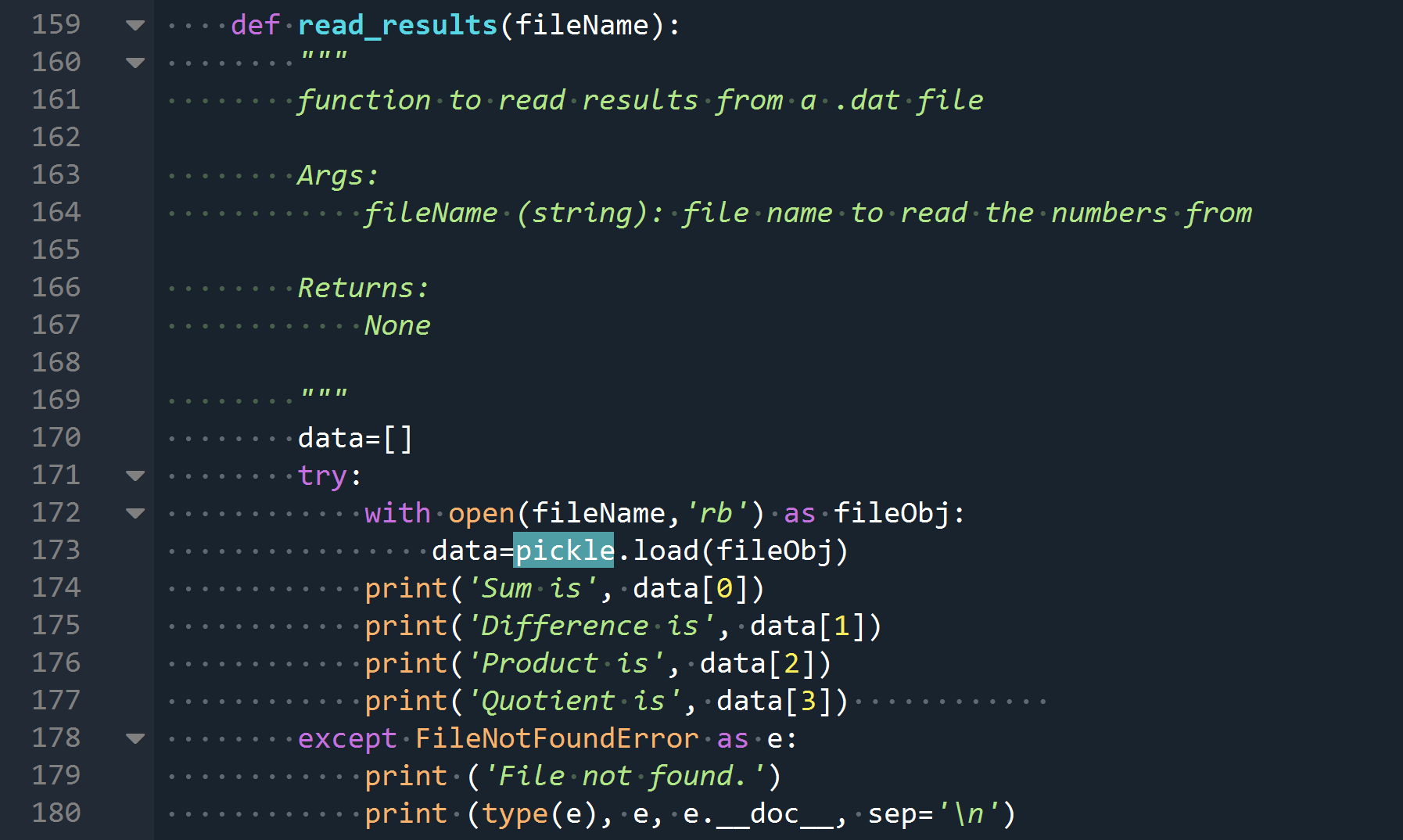


Figure 13 - Unpickling data

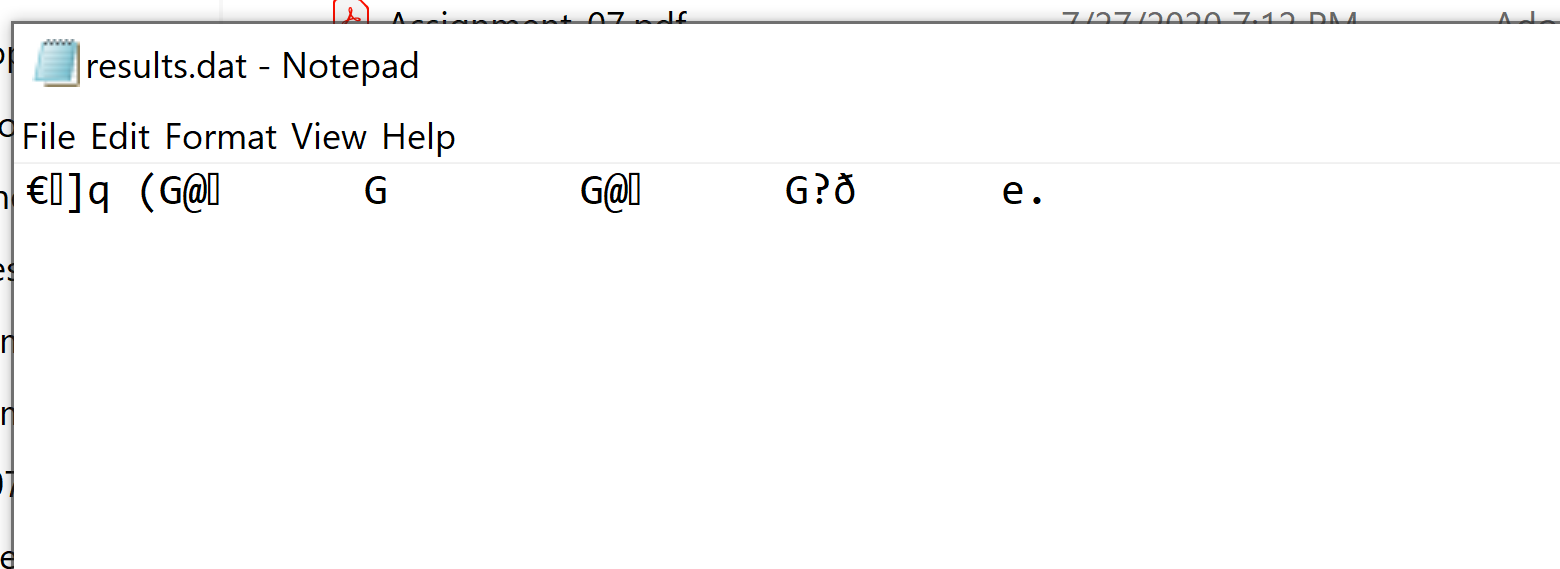


Figure 14 - Pickled data (binary file)

Some Use Cases of Using Pickle Method

I found two interesting uses of pickling method in python in the real-world. It came as no surprise that pickling is useful for data scientists.

First example I found a data scientist who was trying find an answer to this question:

“*Is there any relationship between the GDP (in terms of purchasing power parity) of a country and the percentage of its Internet users? And is this trend similar for low-income/middle-income/high-income countries?”*

Now this data scientist found that he can gather this information by web scraping the CIA (yes *the* CIA) World Fact book [webpage](https://www.cia.gov/library/publications/the-world-factbook/geos/us.html)[[4]](#footnote-4). The scientist used a variety of python modules to build his database and visualizations.

He used **Beautiful Soup** to ‘crawl’ the webpage and download all the text data into a dictionary. And this is where pickling comes in! He pickles this dictionary data so that he can just reload it in his Jupyter notebook without having to web craw/scrape the pages again. Check out this [website](https://towardsdatascience.com/data-analytics-with-python-by-web-scraping-illustration-with-cia-world-factbook-abbdaa687a84)[[5]](#footnote-5) for the full story.

For the second example, another data scientist used it to preserve the data model she has trained, fit and tested to predict on new data at a later time. It was particularly useful for this data scientist in situations were *training* a model could take hours. Instead of having to retrain the model every time it is needed, the model can be pickled for later use and then unload it later for immediate use. Please check out this [webpage](https://medium.com/swlh/pickling-in-python-ac3c7a045ae5)[[6]](#footnote-6) for the full story.

Drawbacks of Pickling

Unfortunately, pickled data is not secure and can be hacked. It can be used to execute arbitrary code during the unpickling process. There isn’t much that can be done to reduce this risk. So the rule of thumb to follow is to never unpickle data from an untrusted source or is transmitted over an insecure network. In order to prevent man-in-the-middle attacks, it’s a good idea to use libraries such as **hmac** to sign the data and ensure it hasn’t been tampered with.[[7]](#footnote-7)

# Structured Error Handling

Back again to the world of error handling. I have dabbled a little bit on error handling in the previous assignment but in this homework, I learn how to customize exceptions so that it catches specific exceptions.

Initially, in the previous homework, I used the exception handling to catch a failure in a custom that is used for reading files. Observe Figure 15 - Basic Exception Handling with try and except statements. In Line 99, I put the **try** keyword before the statement for opening my text file. And then on Line 108, under the **except** keyword, I have a print statement that would trigger if opening the file fails. This code works and it does capture when an exception occurs during the opening of the file. **But it is good programming practice to specify exception types so that you handle each individual case. Generally, a catchall exception should be avoided.**

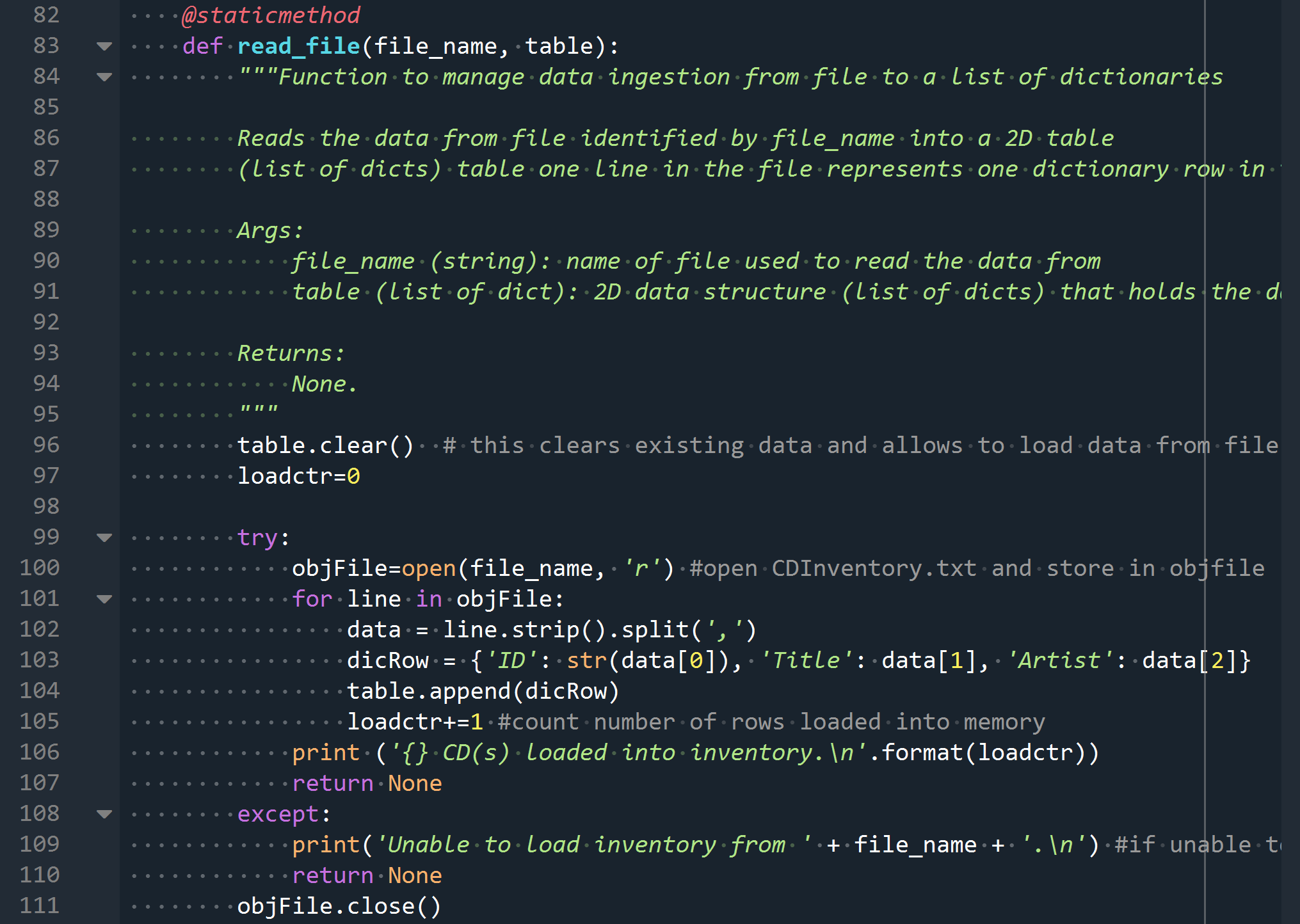


Figure 15 - Basic Exception Handling with try and except statements

So I can further improve me exception error handling by catching the exception type concerning a file not found issue. To find out what the specific exception type is for my error, I purposely generated the error by trying to open a file that does not exist. In Figure 15 - Basic Exception Handling with try and except statements, you can see in the last second to the last line says: **FileNotFoundError: [Errno2]**. So, this is the clue I need. I have to catch FileNotFoundError exception type.

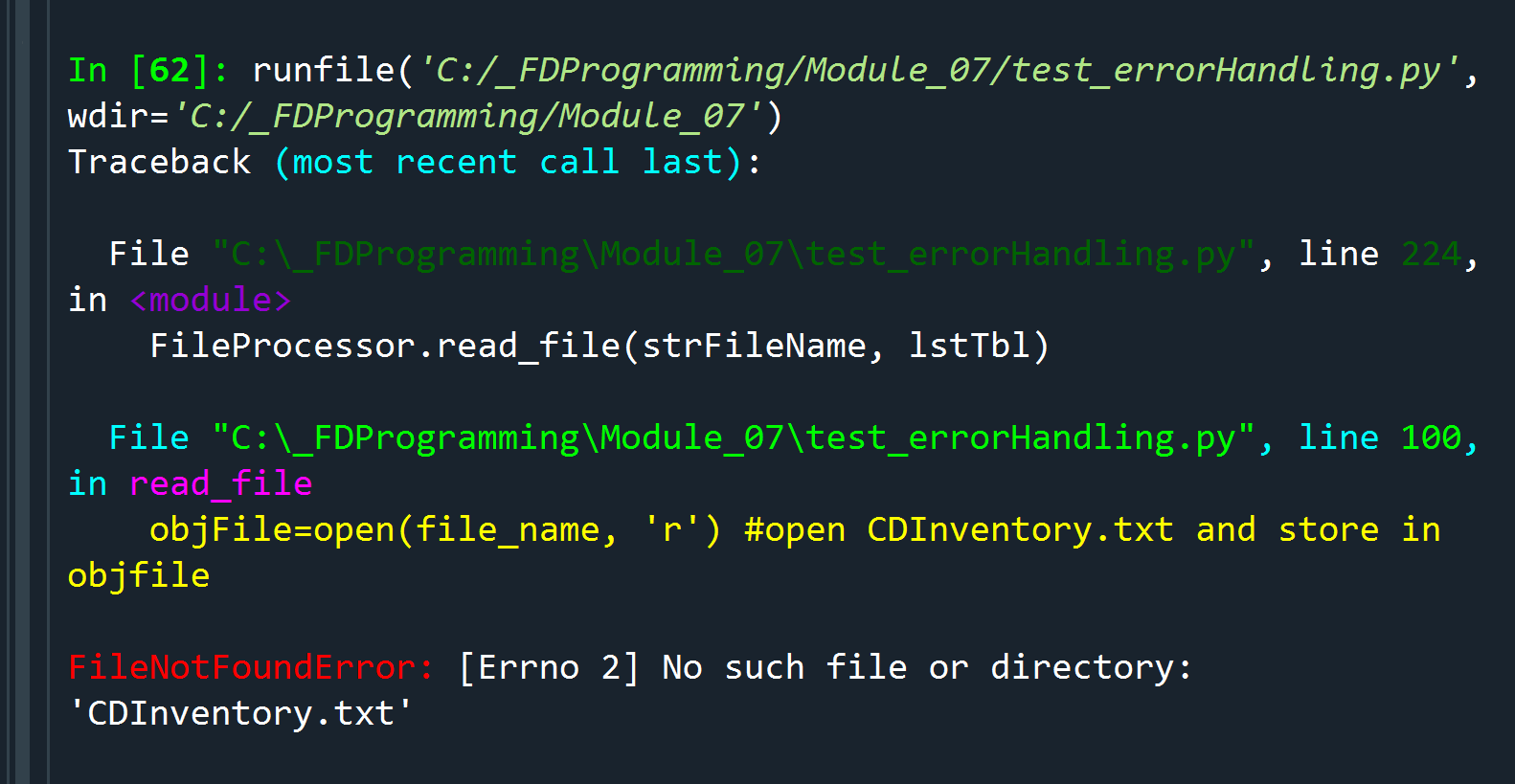


Figure 16 - FileNotFoundError

This is exactly what I did in Figure 17 - Exception Type FileNotFoundError. So in the except statement at Line 154, I specifically am capturing the FileNotFoundError and storing it in the **e** variable.

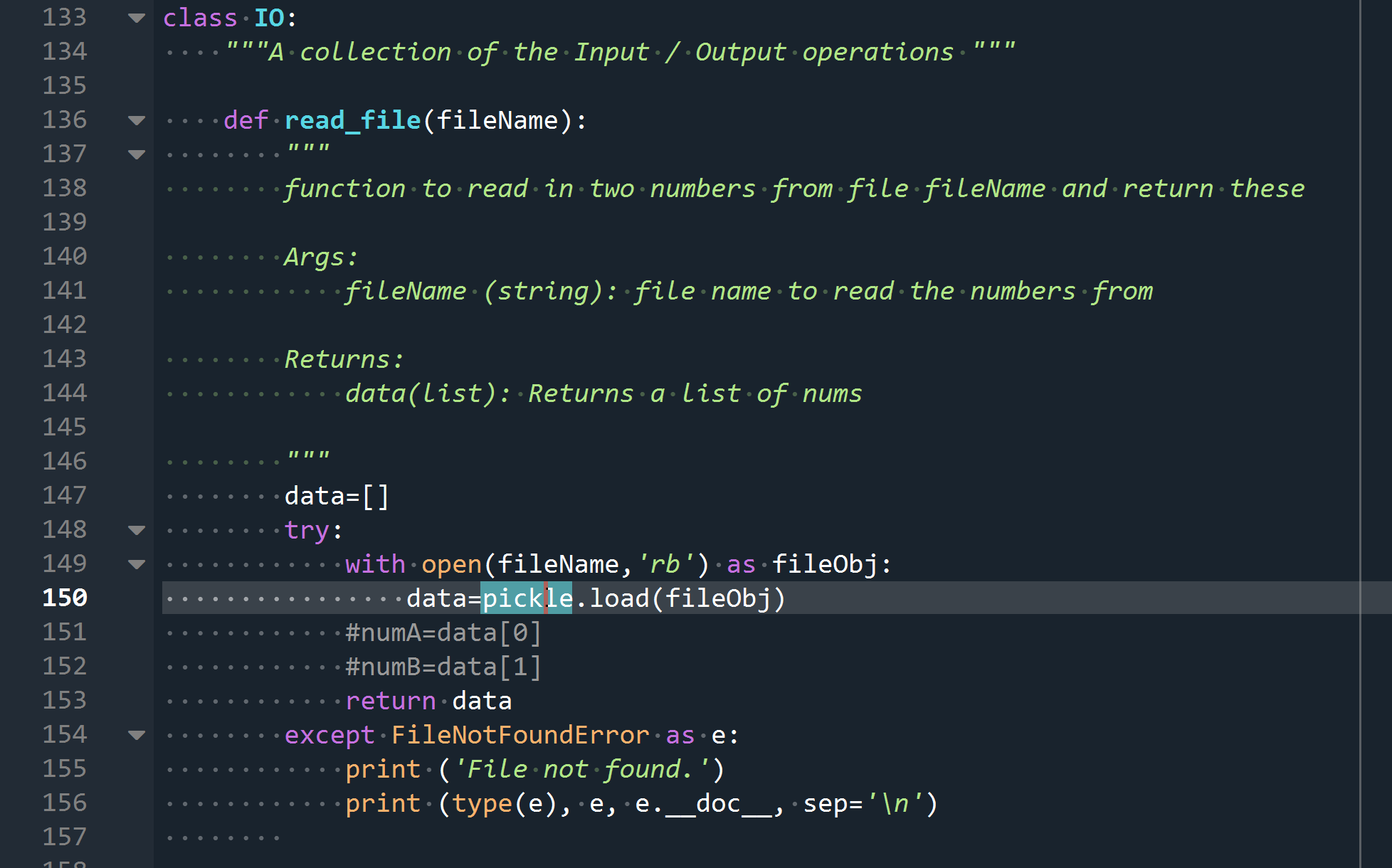


Figure 17 - Exception Type FileNotFoundError

In Figure 18 - Lab07\_C Code Output, I purposely generated an error so that it can be caught by my **try** and **except** clauses. As you can see, my custom message ‘*File not found.*’ is printed out as well as what the exception class was caught; in my case it is the ‘*FileNotFoundError’*. Finally, I also printed out the **\_\_doc\_\_ string** pertaining to my error: *[Errorno 2] No such file or directory: ‘results.dat’*. This last bit is really helpful because it even tells me what file was not found which is results.dat.

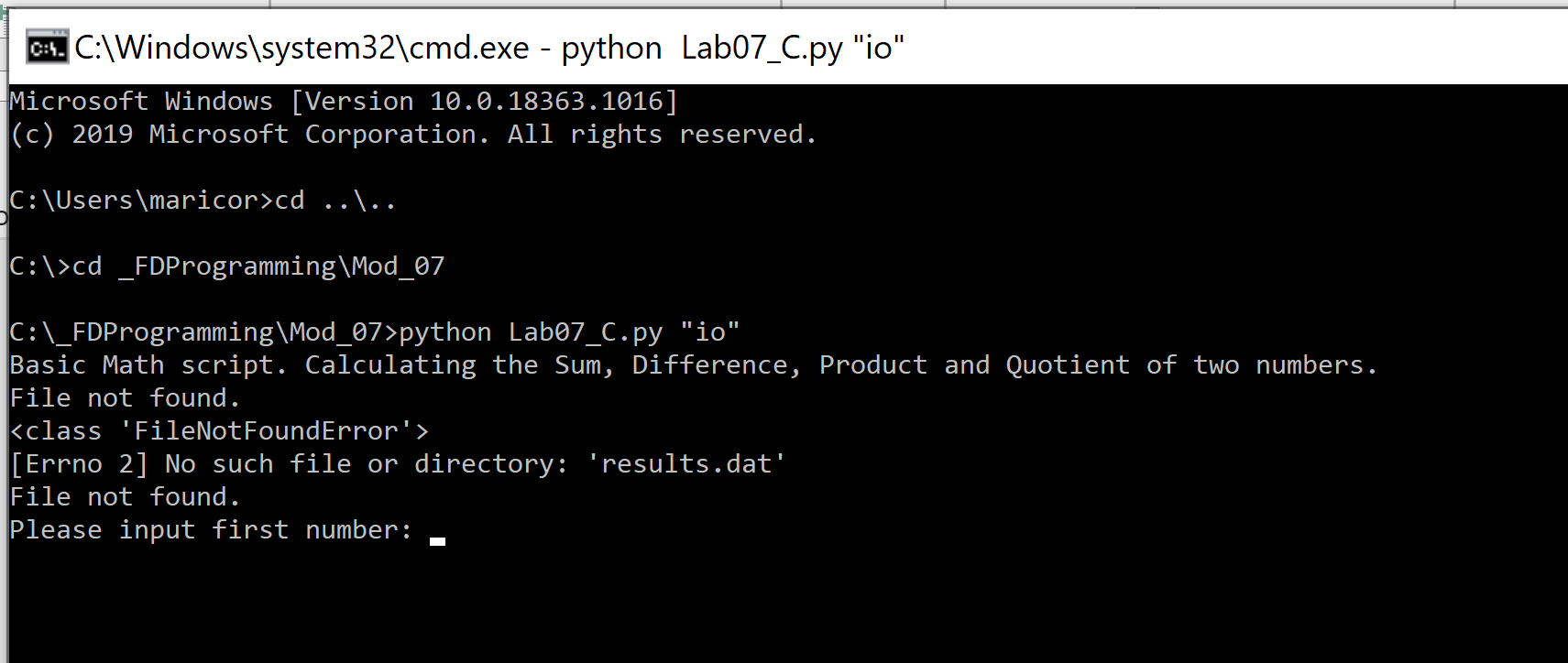


Figure 18 - Lab07\_C Code Output

For a complete list of **built-in python exceptions**, you may refer to this [link](https://docs.python.org/3/library/exceptions.html)[[8]](#footnote-8). This website also shows the exception hierarchy. In my specific example, FileNotFoundError, this belongs under the **OSError** parent class.

I was curious what **[Errno 2]** is from my FileNotFound exception error. While researching, these two commands below can list out all the errorcodes. Based on that, I found that Errno 2 refers to '**ENOENT**'.

**import errno**

**print (errno.errorcode)**

To find out what ENOENT is, I found this [link](https://docs.python.org/2/library/errno.html)[[9]](#footnote-9) very useful in deciphering its meaning. This is shown in Figure 19 - ENOENT Meaning

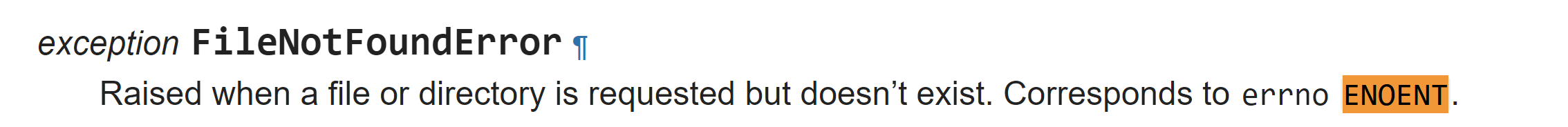


Figure 19 - ENOENT Meaning

Exception Handling Examples

I have already discussed one very common use case where exception handling is a good idea, this is when trying to open files that does not exist. Other scenarios that I can see that I might possible run into is permission issues (errno.**EACCES**) or not having enough disk space (errno.**ENOSPC**) to write file(s) to. Another scenario where exception handling can be useful is in where data is being loaded from an empty file. That in itself may not cause an error but say for example that the null data is being assigned to lists, it can potentially lead to ‘index out of range’ exception. As for unpickling files that are empty, this can lead to EOFError errors so is another scenario where exception handling should be included.

# CDInventory Program

For the latest version of CDInventory.py, I employed exception handling when a file that does not exist is being loaded into memory. I also added exception handling in case user does not have permission to create or modify a file as well as not having enough disk space. There is also exception handling for pickling and unpickling errors as well as unpickling an empty file (EOFError).

I also pickled my entire 2D table and stored it as CDInventory.day and then unpickled it when the ‘load’ option is selected by user.

Please observe Figure 20 - CDInventory.py in Spyder as well as Figure 21 - CDInventory.py in Windows Console.

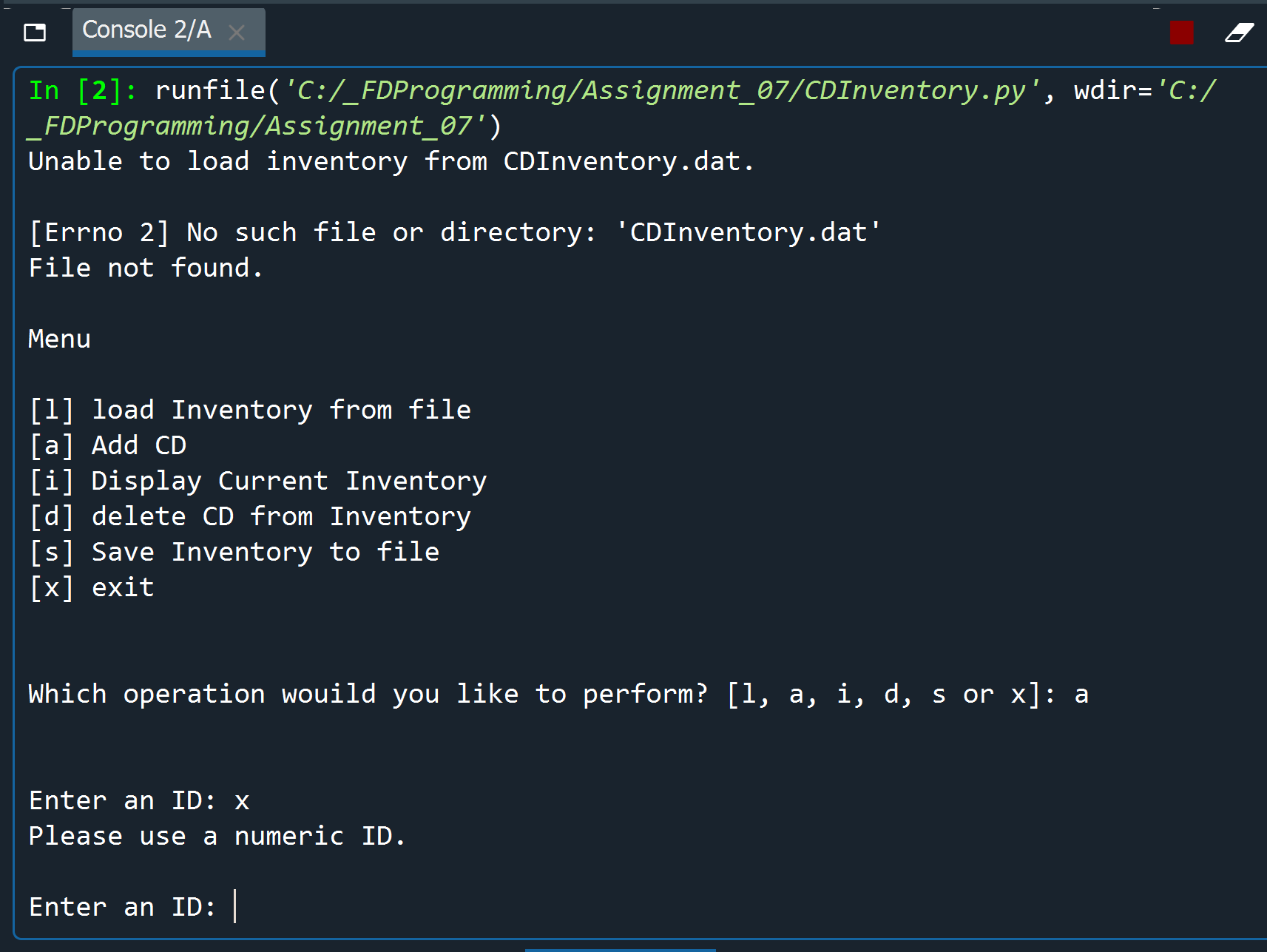


Figure 20 - CDInventory.py in Spyder

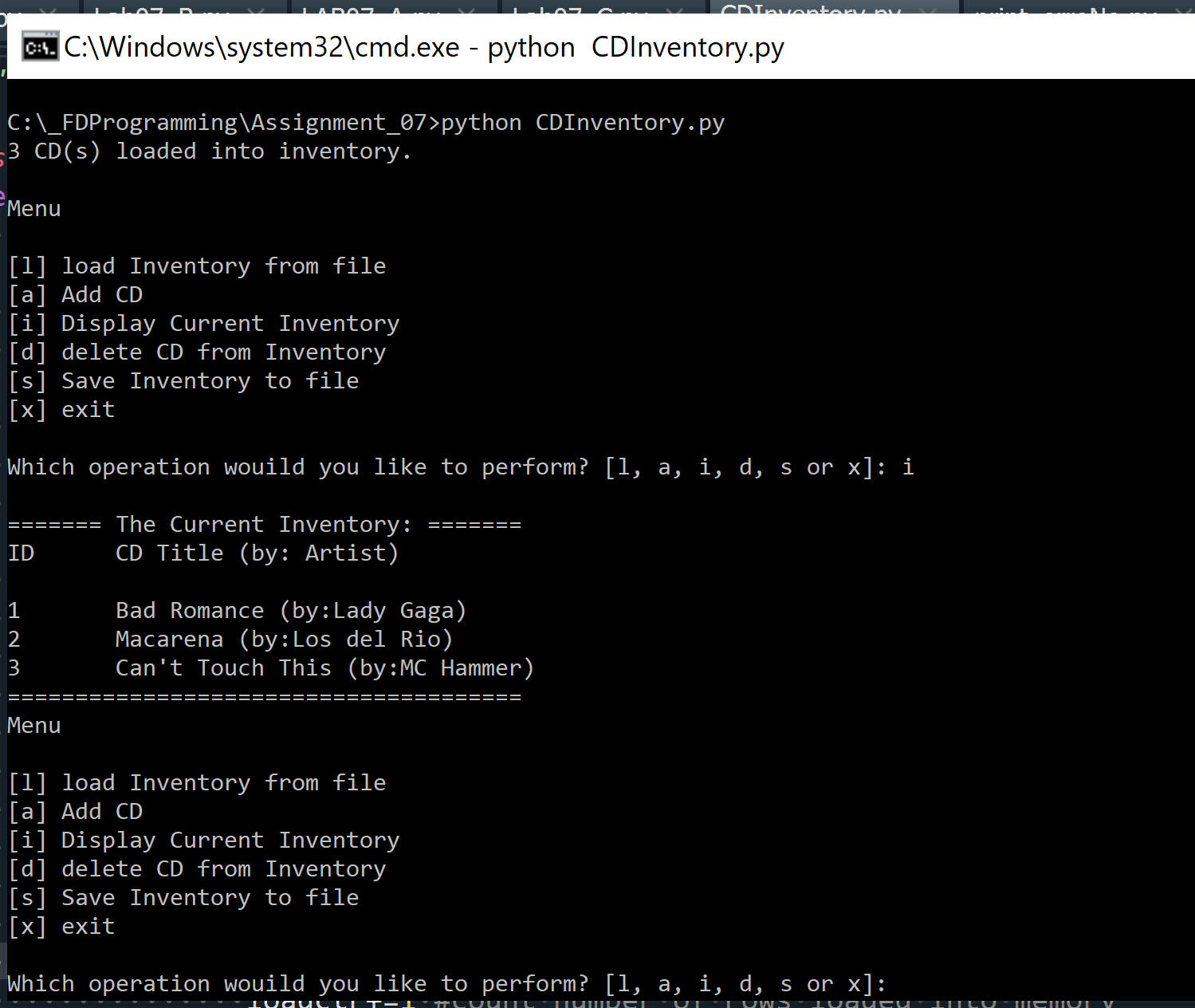


Figure 21 - CDInventory.py in Windows Console

# GitHub Flavored Markdown

I am introduced about m**arkdown** language in GitHub. **Markdown** is a lightweight and easy-to-use syntax for styling all forms of writing on the GitHub platform. In the short intro I was exposed too, having knowledge of markdown can help me stylize documents such as the readme doc in GitHub.

# Summary

I learned a lot of notable things in Module 7. First of which is how to pickle and unpickle data plus the security concerns that comes with it. As a rule of thumb, never unpickle data from an untrusted source or transmit over an unsecured connection. I dug deeper in the world of structured error handling. I have gained the importance of being as specific as I can when catching exceptions and try and avoid a catchall exception handling. This [website](https://www.codementor.io/@henrygeorge/a-comprehensive-guide-to-handling-exceptions-in-python-178ikwc2ot)[[10]](#footnote-10)has a lot of good suggestions/best practices when it comes to exception handling such as not using sensitive information in your exception messages or consistency with the strictness of your code.

Lastly, I have a brief intro to using markdown language in Github as a way of customizing or styling my presentation of the readme doc.

## Appendix

1. **Lab07\_C.py**
2. #------------------------------------------#
3. # Title: LAB06\_C.py
4. # Desc: simple demonstrator for classes
5. # Change Log: (Who, When, What)
6. # DBiesinger, 2030-Jan-01, Created File
7. #------------------------------------------#
9. **import** pickle
10. **import** sys
12. # -- DATA -- #
13. strFileInput = 'numbers.dat'
14. strFileOutput = 'results.dat'
15. intNumA=0
16. intNumB=0
18. # -- PROCESSING -- #
19. **class** SimpleMath:
20. """A collection of simple math processing functions """
22. @staticmethod
23. **def** get\_sum(val1 = 0.0, val2 = 0.0):
24. """Function for adding two values

27. Args:
28. val1: the first number to add
29. val2: the second number to add

32. Returns:
33. A float corresponding to the sum of val1 and val2
34. """
35. addSum=0.0
36. **try**:
37. addSum=float(int(val1) + int(val2))
38. **return** addSum
39. **except** ValueError as e:
40. **print** ('val1 is', val1)
41. **print** ('vale2 is', val2)
42. **print** ('Found non-integer operands!')
43. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
44. **return** addSum

47. @staticmethod
48. **def** get\_diffference(val1 = 0.0, val2 = 0.0):
49. """Function for subtracting two values

52. Args:
53. val1: the number to subtract from
54. val2: the number to subtract

57. Returns:
58. A float corresponding to the difference of val1 and val2
59. """
60. diff=0.0
61. **try**:
62. diff=float(int(val1) - int(val2))
63. **return** diff
64. **except** ValueError as e:
65. **print** ('Found non-integer operands!')
66. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
67. **return** diff
68. **except** TypeError as e:
69. **print** ('val1 is', val1)
70. **print** ('val2 is', val2)
71. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
72. **return** diff
74. @staticmethod
75. **def** get\_product(val1 = 0.0, val2 = 0.0):
76. """Function for multiplying two values

79. Args:
80. val1: the first number to multiply
81. val2: the second number to multiply

84. Returns:
85. A float corresponding to the product of val1 and val2
86. """
87. prod=0.0
88. **try**:
89. prod=float(int(val1) \* int(val2))
90. **return** prod
91. **except** ValueError as e:
92. **print** ('Found non-integer operands!')
93. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
94. **return** prod
95. **except** TypeError as e:
96. **print** ('val1 is', val1)
97. **print** ('val2 is', val2)
98. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
99. **return** prod
101. @staticmethod
102. **def** get\_quotient(val1 = 0.0, val2 = 0.0):
103. """Function for dividing two values

106. Args:
107. val1: the number to divide
108. val2: the number to divide by

111. Returns:
112. A float corresponding to the quotient of val1 and val2
113. """
114. quotient=0.0
115. **try**:
116. quotient = float(int(val1) / int(val2))
117. **return** quotient
118. **except** ZeroDivisionError as e:
119. **print** ('Cannot divide by zero.')
120. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
121. **return** quotient
122. **except** ValueError as e:
123. **print** ('Found non-integer operands!')
124. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
125. **return** quotient
126. **except** TypeError as e:
127. **print** ('val1 is', val1)
128. **print** ('val2 is', val2)
129. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
130. **return** quotient


134. **class** IO:
135. """A collection of the Input / Output operations """
137. **def** read\_file(fileName):
138. """
139. function to read in two numbers from file fileName and return these
141. Args:
142. fileName (string): file name to read the numbers from
144. Returns:
145. data(list): Returns a list of nums
147. """
148. data=[]
149. **try**:
150. with open(fileName,'rb') as fileObj:
151. data=pickle.load(fileObj)
152. #numA=data[0]
153. #numB=data[1]
154. **return** data
155. **except** FileNotFoundError as e:
156. **print** ('File not found.')
157. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')

160. **def** read\_results(fileName):
161. """
162. function to read results from a .dat file
164. Args:
165. fileName (string): file name to read the numbers from
167. Returns:
168. None
170. """
171. data=[]
172. **try**:
173. with open(fileName,'rb') as fileObj:
174. data=pickle.load(fileObj)
175. **print**('Sum is', data[0])
176. **print**('Difference is', data[1])
177. **print**('Product is', data[2])
178. **print**('Quotient is', data[3])
179. **except** FileNotFoundError as e:
180. **print** ('File not found.')
181. **print** (type(e), e, e.\_\_doc\_\_, sep='\n')
183. **def** write\_results(fileName, results):
184. """
185. function to write the math results to file fileName
187. Args:
188. fileName (string): file Name to write the results to.
189. results (list): The results
191. Returns:
192. None.
194. """
195. with open(fileName,'wb') as fileObj:
196. #print ('results is ', results)
197. pickle.dump(results, fileObj)


201. # -- PRESENTATION (Input/Output) -- #
202. **print**('Basic Math script. Calculating the Sum, Difference, Product and Quotient of two numbers.')
203. strArg=sys.argv[1].lower()
204. **if** (strArg =='io'):
205. IO.read\_results(strFileOutput)
206. lstNum=[]
207. intNumA=input('Please input first number: ')
208. intNumB=input('Please input second number: ')
209. lstNum.append(intNumA)
210. lstNum.append(intNumB)
211. **print** ('lstNum is ', lstNum)
212. IO.write\_results(strFileInput,lstNum)
213. input('Press enter to exit.')
215. **elif** (strArg=='calc'):
216. lstNum=[]
217. lstNum=IO.read\_file(strFileInput)
218. **print** ('lstNum in calc is', lstNum)
219. lstResults = []
220. **if** (lstNum):
221. intNumA=(lstNum[0])
222. intNumB=(lstNum[1])
223. **print** ('intNumA is ', intNumA)
224. **print** ('intNumB is ', intNumB)
225. lstResults.append(SimpleMath.get\_sum(intNumA, intNumB))
226. **print** ('first lstResults after get\_sum is', lstResults)
227. lstResults.append(SimpleMath.get\_diffference(intNumA, intNumB))
228. lstResults.append(SimpleMath.get\_product(intNumA, intNumB))
229. lstResults.append(SimpleMath.get\_quotient(intNumA, intNumB))
230. IO.write\_results(strFileOutput, lstResults)
231. **else**:
232. **print** ('No input data found.')
233. #intNumA, intNumB = IO.read\_file(strFileInput)
234. **CDInventory.py**
235. #------------------------------------------#
236. # Title: Assignment06.py
237. # Desc: Working with classes and functions.
238. # Change Log: (Who, When, What)
239. # Maria Dacutanan, 2020-Aug-16, Updated read\_file function to include error handling for file not existing
240. # Maria Dacutanan, 2020-Aug-16, Updated show\_inventory function to include Check for empty table
241. # Maria Dacutanan, 2020-Aug-16, Added get\_newInventory function in class IO
242. # Maria Dacutanan, 2020-Aug-16, Added add\_newInventory function in class DataProcessor
243. # Maria Dacutanan, 2020-Aug-16, Added code for write\_file function in class FileProcessor
244. # Maria Dacutanan, 2020-Aug-16, Added del\_inventory function in class DataProcessor
245. # Maria Dacutanan, 2020-Aug-18, Updated del\_inventory function to delete duplicate entries
246. #------------------------------------------#
248. # -- DATA -- #
249. strChoice = '' # User input
250. lstTbl = []  # list of lists to hold data
251. dicRow = {}  # list of data row
252. strFileName = '\_CDInventory.txt'  # data storage file
253. objFile = None  # file object
254. loadErr=False
256. # -- PROCESSING --  #
257. **class** DataProcessor:
259. """Add or Delete Data from Inventory"""
261. @staticmethod
262. **def** add\_newInventory(id, title, artist):
263. """Function to add new data into CDInventory
265. Args:
266. id(string): id of new entry
267. title(string): CD title of new entry
268. artist(string): arist's name of new entry
270. Returns:
271. None
272. """
273. dicRow = {'ID': id, 'Title': title, 'Artist': artist}
274. lstTbl.append(dicRow)
275. **return** None
277. @staticmethod
278. **def** del\_inventory(id):
279. """Function to Delete from CDInventory
281. Args:
282. id(string)=id of entry in CDInventory that is to be deleted
284. Returns:
285. None
286. """
288. blnCDRemoved = False
289. lstID=[]
290. delctr=0
291. **for** cd **in** lstTbl:
292. **for** row **in** cd['ID']:
293. lstID.append(row) #Store all IDs from lstTbl into lstID table
294. **if** lstID.count(id) > 0: #Check if user input exists in lstID
295. intRowNr = 0
296. #This while block will loop thru lstTbl to delete ALL instances of ID in case of duplicates
297. **while** intRowNr < len(lstTbl):
298. **if** (lstTbl[intRowNr]['ID']) == id:
299. **del** lstTbl[intRowNr]
300. delctr+=1 #Count number of deletions
301. intRowNr=0 #if ID was deleted, restart intRowNr as lstTbl has shifted
302. **else**:
303. intRowNr += 1#increase intRowNr to move on to next index of lstTbl
304. blnCDRemoved = True
306. **if** blnCDRemoved:
307. **print**('{} CD(s) removed.\n'.format(delctr))
308. **else**:
309. **print**('Could not find this CD!\n')
310. **return** None

313. **class** FileProcessor:
314. """Processing the data to and from text file"""
316. @staticmethod
317. **def** read\_file(file\_name, table):
318. """Function to manage data ingestion from file to a list of dictionaries
320. Reads the data from file identified by file\_name into a 2D table
321. (list of dicts) table one line in the file represents one dictionary row in table.
323. Args:
324. file\_name (string): name of file used to read the data from
325. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime
327. Returns:
328. None.
329. """
330. table.clear()  # this clears existing data and allows to load data from file
331. loadctr=0
333. **try**:
334. objFile=open(file\_name, 'r') #open CDInventory.txt and store in objfile
335. **for** line **in** objFile:
336. data = line.strip().split(',')
337. dicRow = {'ID': str(data[0]), 'Title': data[1], 'Artist': data[2]}
338. table.append(dicRow)
339. loadctr+=1 #count number of rows loaded into memory
340. **print** ('{} CD(s) loaded into inventory.\n'.format(loadctr))
341. **return** None
342. **except**:
343. **print**('Unable to load inventory from ' + file\_name + '.\n') #if unable to load file, return error msg and break out of loop
344. **return** None
345. objFile.close()
347. @staticmethod
348. **def** write\_file(file\_name, table):
349. """Function to Save CDInventory into File
351. Args:
352. file\_name(file object)=filename of CDInventory file
353. table(list)= list of CDInventory dictionaries
355. Return:
356. None
357. """
359. savectr=0
360. objFile = open(file\_name, 'w')
361. **for** row **in** table:
362. lstValues = list(row.values())
363. objFile.write(','.join(lstValues) + '\n')
364. savectr+=1 #counts number of rows saved into file
365. objFile.close()
366. **print** ('{} CD(s) saved into {}.\n'.format(savectr,file\_name))
367. **return** None

370. # -- PRESENTATION (Input/Output) -- #
372. **class** IO:
373. """Handling Input / Output"""
375. @staticmethod
376. **def** print\_menu():
377. """Displays a menu of choices to the user
379. Args:
380. None.
382. Returns:
383. None.
384. """
386. **print**('Menu\n\n[l] load Inventory from file\n[a] Add CD\n[i] Display Current Inventory')
387. **print**('[d] delete CD from Inventory\n[s] Save Inventory to file\n[x] exit\n')
389. @staticmethod
390. **def** menu\_choice():
391. """Gets user input for menu selection
393. Args:
394. None.
396. Returns:
397. choice (string): a lower case sting of the users input out of the choices l, a, i, d, s or x
399. """
400. choice = ' '
401. **while** choice **not** **in** ['l', 'a', 'i', 'd', 's', 'x']:
402. choice = input('Which operation would you like to perform? [l, a, i, d, s or x]: ').lower().strip()
403. **print**()  # Add extra space for layout
404. **return** choice
406. @staticmethod
407. **def** show\_inventory(table):
408. """Displays current inventory table

411. Args:
412. table (list of dict): 2D data structure (list of dicts) that holds the data during runtime.
414. Returns:
415. None.
417. """
418. **if** (table):
419. **print**('======= The Current Inventory: =======')
420. **print**('ID\tCD Title (by: Artist)\n')
421. **for** row **in** table:
422. **print**('{}\t{} (by:{})'.format(\*row.values()))
423. **print**('======================================')
424. **else**:
425. **print** ('Inventory is empty.\n')
426. **return** None
428. @staticmethod
429. **def** get\_newInventory():
430. """Prompts User to provide ID, Title and Arist Name

433. Args:
434. None
436. Returns:
437. strID (string) - ID
438. strTitle (string) - Title
439. strArtist (string) - Artist
441. """
443. **while** True: #user is re-prompted for null ID
444. strID = str(input('Enter an ID: ').strip())
445. **if** (strID):
446. **break**
447. **while** True: #user is re-prompted for null CD Title
448. strTitle = input('Enter the CD\'s Title: ').strip()
449. **if** (strTitle):
450. **break**
451. **while** True: #user is re-prompted for null Artist's Name
452. strArtist = input('Enter the Artist\'s Name: ').strip()
453. **if** (strArtist):
454. **break**
455. **return** (strID, strTitle, strArtist)
457. # 1. When program starts, read in the currently saved Inventory
458. FileProcessor.read\_file(strFileName, lstTbl)
460. # 2. start main loop
461. **while** True:
462. # 2.1 Display Menu to user and get choice
463. IO.print\_menu()
464. strChoice = IO.menu\_choice()
466. # 3. Process menu selection
467. # 3.1 process exit first
468. **if** strChoice == 'x':
469. **break**
470. # 3.2 procless load inventory
471. **if** strChoice == 'l':
472. **print**('WARNING: If you continue, all unsaved data will be lost and the Inventory re-loaded from file.')
473. strYesNo = input('type \'yes\' to continue and reload from file. otherwise reload will be canceled: ')
474. **if** strYesNo.lower() == 'yes':
475. **print**('reloading...')
476. FileProcessor.read\_file(strFileName, lstTbl) # function call to read CDInventory.txt
477. IO.show\_inventory(lstTbl)
478. **else**:
479. input('canceling... Inventory data NOT reloaded. Press [ENTER] to continue to the menu.')
480. IO.show\_inventory(lstTbl)
481. **continue**  # start loop back at top.
483. # 3.3 process add a CD
484. **elif** strChoice == 'a':
485. # 3.3.1 Ask user for new ID, CD Title and Artist
486. intID,strTitle,strArtist=IO.get\_newInventory() #function call to prompt user for ID, CD Title and Artist and unpack return values
487. DataProcessor.add\_newInventory(intID, strTitle, strArtist) #function call to add data into inventory
488. IO.show\_inventory(lstTbl)
489. **continue**  # start loop back at top.
490. # 3.4 process display current inventory
491. **elif** strChoice == 'i':
492. IO.show\_inventory(lstTbl)
493. **continue**  # start loop back at top.
494. # 3.5 process delete a CD
495. **elif** strChoice == 'd':
496. # 3.5.1 get user input for which CD to delete
497. # 3.5.1.1 display Inventory to user
498. **if** (lstTbl): #check if lstTbl is not empty
499. # 3.5.1.2 ask user which ID to remove
500. **while** True:
501. intIDDel = input('Which ID would you like to delete? ').strip()
502. **if** (intIDDel): #user is re-prompted for empty ID
503. DataProcessor.del\_inventory(intIDDel) #function call to delete user provided ID
504. **break**
505. IO.show\_inventory(lstTbl)
506. **else**:
507. **print**('Nothing to delete. Inventory is empty.\n')
508. **continue**  # start loop back at top.
509. # 3.6 process save inventory to file
510. **elif** strChoice == 's':
511. # 3.6.1 Display current inventory and ask user for confirmation to save
512. **if** (lstTbl):
513. IO.show\_inventory(lstTbl)
514. strYesNo = input('Save this inventory to file? [y/n] ').strip().lower()
515. # 3.6.2 Process choice
516. **if** strYesNo == 'y':
517. # 3.6.2.1 save data
518. FileProcessor.write\_file(strFileName, lstTbl) #function call to write inventory into file
519. **else**:
520. input('The inventory was NOT saved to file. Press [ENTER] to return to the menu.')
521. **else**:
522. **print**('Nothing to save. Inventory is empty.\n')
523. **continue**  # start loop back at top.
524. # 3.7 catch-all should not be possible, as user choice gets vetted in IO, but to be safe:
525. **else**:
526. **print**('General Error')

1. Last retrieved 23-Aug-20 [↑](#footnote-ref-1)
2. Last retrieced from [wikipedia](https://en.wikipedia.org/wiki/Serialization#:~:text=In%20computing%2C%20serialization%20(US%20spelling,later%20(possibly%20in%20a%20different) on 24-Aug-20 [↑](#footnote-ref-2)
3. Last retrieve from this [link](https://www.tutorialspoint.com/object_oriented_python/object_oriented_python_serialization.htm#:~:text=In%20the%20context%20of%20data,or%20transmitted%20and%20reconstructed%20later.) on 24-Aug-20 [↑](#footnote-ref-3)
4. Last retrieved 24-Aug-20 [↑](#footnote-ref-4)
5. Last retrieved 24-Aug-20 [↑](#footnote-ref-5)
6. Last retrieved 24-Aug-20 [↑](#footnote-ref-6)
7. Quoted from this [webpage](https://realpython.com/python-pickle-module/). Last retrieved 24-Aug-20 [↑](#footnote-ref-7)
8. Last retrieved 25-Aug-20 [↑](#footnote-ref-8)
9. Last retrieved 25-Aug-20 [↑](#footnote-ref-9)
10. Last retrieved 25-Aug-20 [↑](#footnote-ref-10)