Nabil Knight

2022-Mar-13

IT FDN Programming: Python

Assignment 07

Binary and Text Files and Structured Error Handling

# Introduction:

In the seventh week of the Foundations of Programming course we covered topics relating to various file types, pickling data and structured error handling. We dove into text files again, reading and writing to them, and working with binary files, which is a different kind of file, .dat as opposed to a .txt. We also covered new ways to read and append data using readline/s() and .append. Additionally, we covered how to use for and while loops to iterate over our data from files. We also went over using binary files and importing the pickle module to preserve the data you are reading and writing to. Finally, we went over how to use structured error handling to avoid common errors that might crash your program using the try /except concept in greater detail. To put our learnings into practice we were tasked with updating our week six submission to use a binary file and adding error handling to our script.

# Working with Text Files:

Our first main topic for the week was continuing to work with text files. We briefly covered how to read, write and append data to a .txt file within a function. This is similar to material from previous weeks. However, we began diving into other methods of being able to read files using the read and readline/s() function. Sample code below:

fileObj = open(fileName, 'r')

data = fileObj.readline()

fileObj.close()

**return** data

fileObj = open(fileName, 'r')

data = fileObj.read()

fileObj.close()

**return** data

As you can see above, the syntax is different than in past weeks and using the readline function, you read one line at a time (you can rerun your program over and over and it will read through the next line). If you close your program however, it will start over at index 0. When you couple the readline function with a while loop you can iterate over your data line by line – assuming you have multiple lines to read. Or you can use the readlines function to read through the whole file to return a list, while the read function returns a string.

# Binary Files and the Pickle module:

We began to learn about using binary files, they are not like a .txt file in that the data, while isn’t encrypted, isn’t easily legible. A person can read some data within a file but it is not in a standard format that .txt would be. The benefit of using a binary file is that you don’t need to process or reformat the data to save and read from a file. You access the data in memory and write to a file or read from file. This makes the process more efficient and may potentially save memory usage. To be able to accomplish this we need to use a module called pickle or known as pickling in Python. The pickle module grabs the data and serializes it so that it can be easily stored as binary information. In Lab7B we began using the pickle module to read and write to files. The syntax to use this is a lot easier than we’ve been previously using with text files to read and write data. Example below:

**with** open(fileName, 'wb') **as** file:

pickle.dump(data, file)

**with** open(fileName, 'rb') **as** file:

data = pickle.load(file)

**return** data

We also had to use system arguments to be able to call each of the portions of code and to do this we need to create two paths: IO and calc. Code below:

selector = sys.argv[1]

**if** selector == 'IO':

#pass

#path IO

#try

line = IO.read\_file(strFileResults)

print('Results: ', ', '.join(line))

#except:

numA, numB = IO.user\_input()

numbers = [numA, numB]

IO.write\_file(strFileNumbers, numbers)

#display current results out of file

#ask user input (2 numbers)

#save user input to file

**elif** selector == 'calc':

#path calc

numbers = IO.read\_file(strFileNumbers)

To begin with I was very confused with this lab, my first attempt utilized many of the same items but I basically tried to redo all of the code in each of the paths as opposed to creating the paths and then updating the functions as well. Dirk’s walk through of this really helped as this concept was a bit tough for me to grasp. To be able to call the two paths within the terminal, see below:

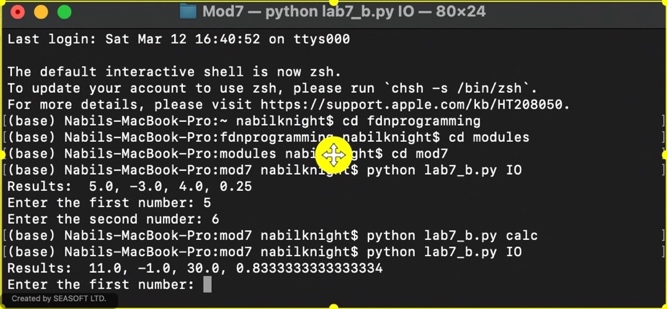


Figure 1 - Lab7 - B Results

We were also tasked with researching various articles that explains what pickling is and I enjoyed the two below:

1. <https://towardsdatascience.com/do-not-use-python-pickle-unless-you-know-all-these-facts-d9e8695b7d43>
2. <https://www.geeksforgeeks.org/understanding-python-pickling-example/>

# Structured Error Handling:

We also began learning about structured error handling with the try / except concept. Typically, you want to use error handling when there is user input, reading and writing to a file, and sections that might have other programmers using your code. The reason for structured error handling is to ensure that your program doesn’t crash while something is engaging with it and potentially losing data / corrupting your file. To do this we use the try / except or exception class. Some typical errors that we have encountered so far in the class are, ValueError, FileNotFound and ZeroDivisionError. These were especially helpful in Lab7C when dealing with users entering integers and trying to open a file that either didn’t exist or didn’t have data within it. When using the exception class in your structured error handling you can extract valuable information about what the error is. This might not be as useful for the customer or user, so as the programmer, you should take this information and find the best way to deliver the right information to them so that they avoid the same error in the future.

**except** ValueError **as** e:

**except** FileNotFound **as** e:

**except** ZeroDivisionError **as** e:

print(type(e), e, e.\_\_doc\_\_, sep='\n')

As you can see above, the programmer sets the exception in variable e so that they can utilize the results in a print statement (or you can create your own custom message). Additionally, you can create your own custom exception classes when you need to have error handling for specific situation that predefined ones don’t solve for or need to replace.

**class** **ValueToLowError**(Exception):

"""Value exceeds lower bound"""

**def** **\_\_str\_\_**(self):

**return** 'Value entered is below allowed lower limits'

The programmer is telling python that they are creating a class and the base class is the exception class. We are supposed to dive into this more next week.

We were tasked with researching exception handling as part of our assignment this week and I thought the below articles did a nice job explaining with various use cases:

1. <https://www.geeksforgeeks.org/python-exception-handling/>
2. https://docs.python.org/3/tutorial/errors.html

# Assignment:

To put our learnings into practice we were supposed to take our week six assignment, write to a binary file and add error handling to our program. As mentioned earlier, to do this we will need to import the pickle module so that we can utilize pickle.load and pickle.dump for our read and write functions as shown below:HIGHLIGHTED CODE

@staticmethod

**def** **read\_file**(file\_name):

"""Function to manage data ingestion from binary file using the pickle module to a list of dictionaries

Reads the data from file identified by file\_name into a 2D table

(list of dicts) table, one line in the file represents one dictionary row in table.

Args:

file\_name (string): name of file used to read the data from

Returns:

table (list of dict): 2D data structure (list of dicts) that holds the data during runtime

"""

**with** open(file\_name, 'rb') **as** objFile:

table = pickle.load(objFile)

**return** table

@staticmethod

**def** **write\_file**(table, file\_name):

"""

Function to save cd inventory from memory to binary file using the pickle module

Args:

file\_name (string): name of file used to read the data from

table (list of dict): 2D data structure (list of dicts) that holds the data during runtime

Returns

-------

None.

"""

**with** open(file\_name, 'wb') **as** objFile:

pickle.dump(table, objFile)

As previously stated, this code is so much simpler than we’ve previously had to use to read, write and format our data into and out of a file. This just takes the list and saves it from memory to file and reads from file in binary format. Our next objective was to write code that would support error handling for our various user inputs and reading and writing data, see below:

**try**:

# 1. When program starts, read in the currently saved Inventory

FileProcessor.read\_file(strFileName)

**except** FileNotFoundError **as** e:

e = 'No such CD Inventory file exists, please enter CD info and save'

print(e + '\n')

#print(type(e), e, \_\_doc\_\_, sep = '|n')

**except** EOFError **as** e:

e = 'Empty File, please enter CD Inventory to be saved'

print(e)

To avoid the error of not having a file or if the file is empty I have added two exceptions to account for this, with their own custom messages to let the user know what they need to do to avoid this in the future. Initiating the program, the exception raised below:

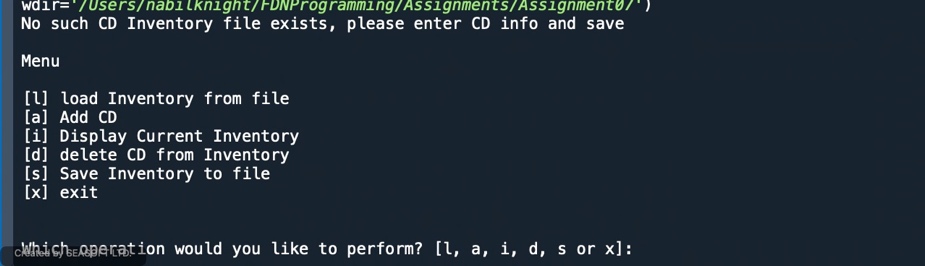


Figure 2 - No File Exception

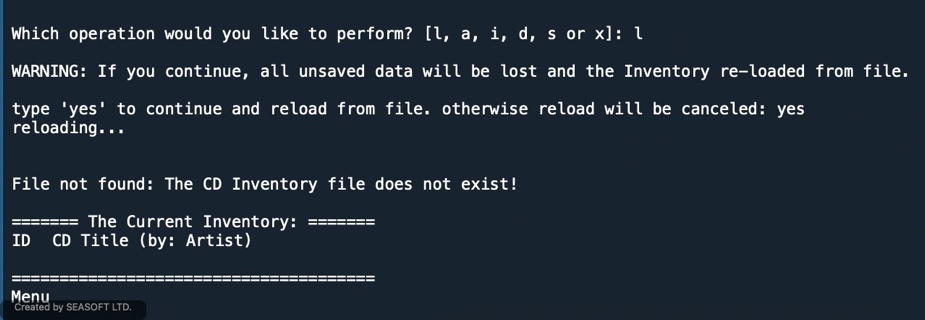


Figure 3 - Loading CD Inventory Exception

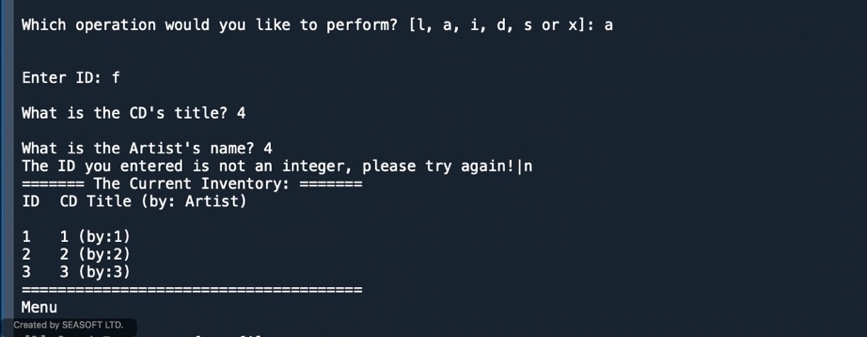
I also included exceptions to handle user input errors when adding and deleting rows of data as seen below:

Figure 4 - Adding entries Exception

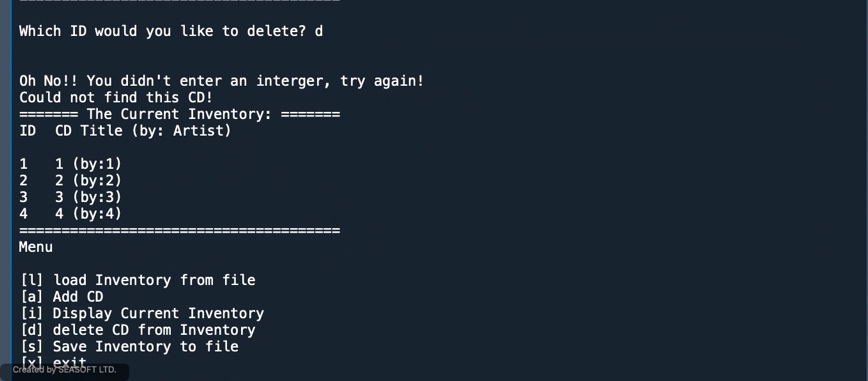


Figure 5 - Deleting Rows Exception

I have included the working results in the appendix for your awareness. I tried opening the results.dat file with textedit and for some reason it couldn’t open, any suggestions (I am using a mac)?

# Conclusion:

In the seventh week of our course we learned about files in more depth, most notably the various ways we can read and write to text files and utilizing binary files. We went over topics such as using loops to iterate over the data we are reading and writing to/from using various functions such as read() and readline/s(). Binary files are simpler to use in that you do not need to do much of the formatting that a text file needs however, you need to import the pickle module to be able to write to these file types. Additionally, we covered structure error handling to be able to ensure your program doesn’t crash and potentially lose valuable data. To put our learnings to practice we updated our week six assignment to include structure error handling and using a binary file.

# Appendix:

# 

Figure 6 - Spyder Results pt. 1

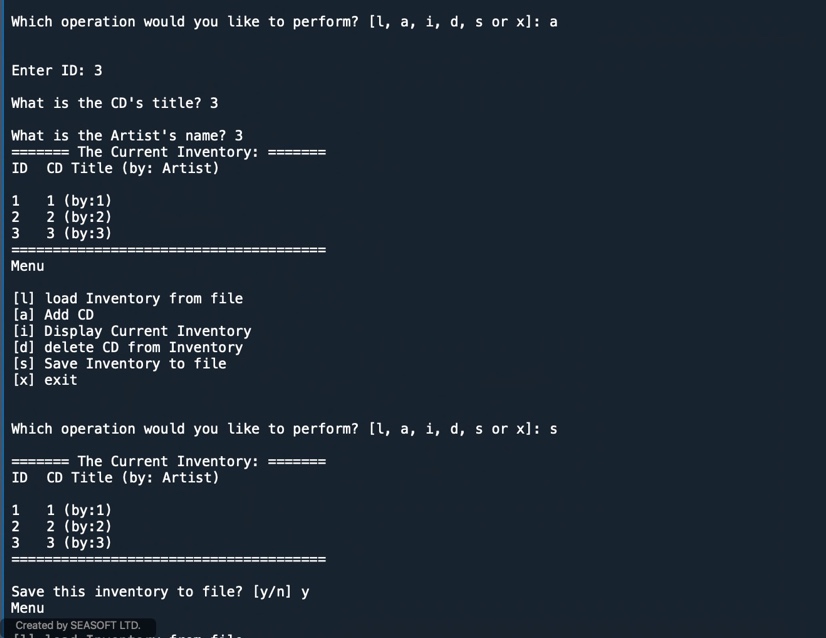


Figure 7 - Spyder Results pt. 2

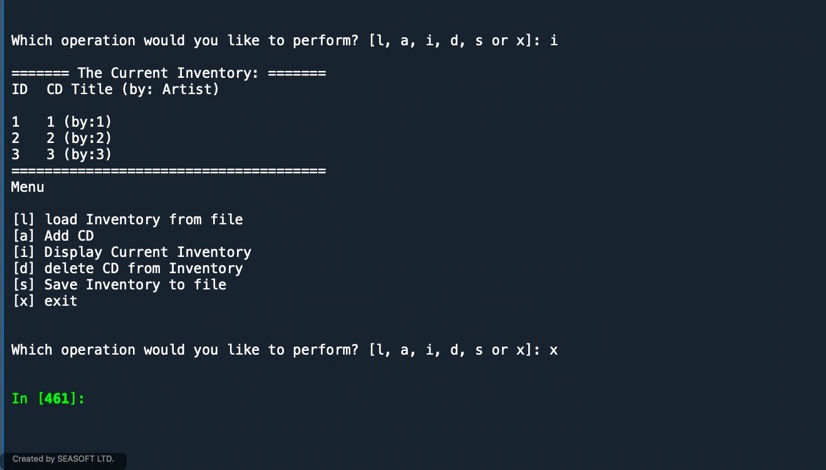


Figure 8 - Spyder Results pt. 3



Figure 9 - Terminal Results pt. 1

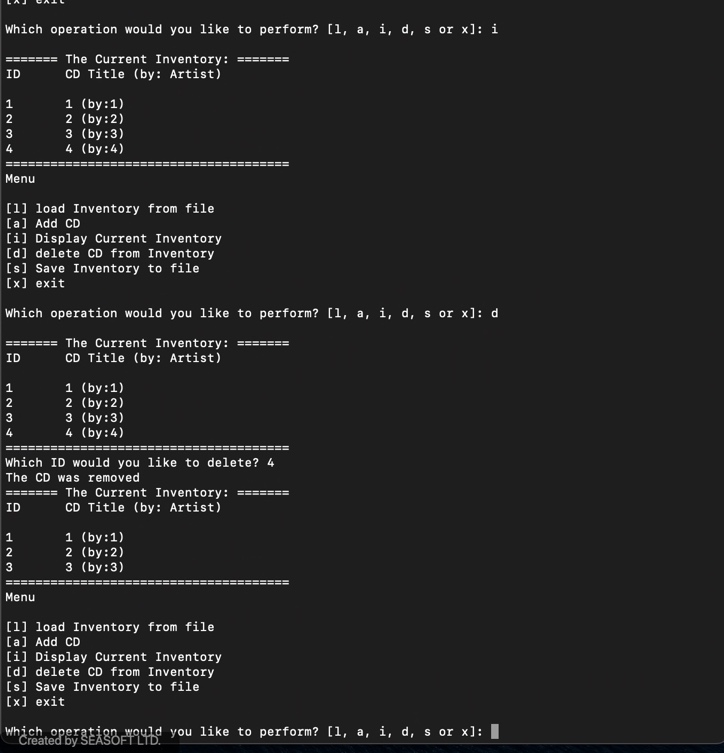


Figure 10 - Terminal Results pt. 2

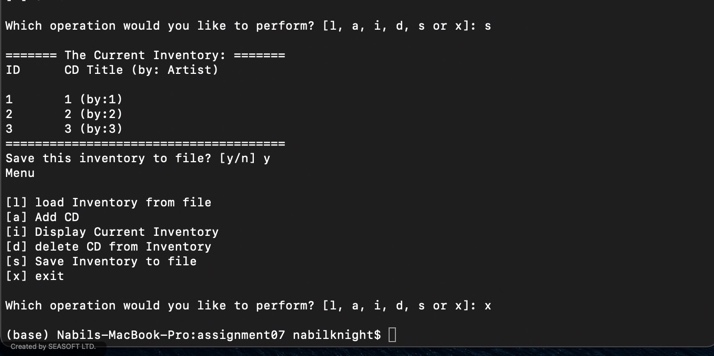


Figure 11 - Terminal Results pt. 3