Name: Matthew Penny

Date: 2021 AUG 20

Course: Foundations Of Programming: Python

Assignment: Assignment 07

**Error Handling, File Operations, and Pickling**

**Introduction**

In this module, we covered the topics of error handling, file operations, and pickling. I covered various study materials including videos and web pages and then worked through three labs before a final project. This module ended up being the most time consuming by a large measure compared to past modules and the final project still contains two noted bugs (explained below) that I was unable to fix after hours of effort. This module was indeed quite informative and hopefully I can discover the cause of unexpected behavior in my updated script.

**Lessons Learned and Concepts of Interest**

As a requirement for this module, I reviewed all videos and content from the module 07 section here: https://saravji.github.io/saravjis\_hut/FDN\_Prog/Modules.html (external site)

I also read chapter seven of “Python Programming, Third Edition – For the Absolute Beginner”, reviewed two web pages that detailed examples of content we were reviewing. After covering these materials, I am able to address points of interest laid out in the course materials:

**What are the benefits of using structured error handling?**

Structured error handling gives us a way to “try” a block of code and then execute an “exception” afterwards if a problem occurs. Normally, if an error handling entry isn’t present when a problem occurs, the program will simply crash and print the exception to the screen. To avoid this, you can raise an exception in relation to a “try” block of code. Not only can you display error messages, you can also affect the flow of code and have it do something like continue processing or direct a user back to a point in the code where they can repeat input. In general, it gives us a lot of flexibility in handling errors of many kinds and we can take different program paths in response instead of the program simply crashing and having to be restarted.

**What are the differences between a text file and a binary file?**

A text file holds data in “plaintext” that is, text that is plain and readable by humans. This is what we would typically encounter when opening a text file and is easy to make sense of. Binary files on the other hand are full of encoded data. This isn’t a plain storage of just text, but also metadata (data about the data) as an object such as what properties it may have. This data is “encoded” – meaning that the characters appear differently than they would in plaintext. The data is, by default, not readable by humans – but the value lies in the added data stored in the file itself.

**How is the Exception class used?**

The exception class is called with the keyword “except” followed by a specified type of exception that was expected to be caught or triggered. For example, the “ZeroDevisionError” is often used in mathematical operations and specified like so:

Try:

numA=10

numB=0

try:

numA/numb

except ZeroDevisionError as e

print(e)

In this case, we’re expecting a ZeroDevisionError to occur in the try block. When it occurs, we raise an exception and store the output in “e” before printing it out. There’s a large number of exception types within the class itself. If one is unsure which to use, the main “exception” class as it’s the main exception type that all other built-in exceptions are derived from.

**How do you "derive" a new class from the Exception class?**

You do that by inheriting from the Exception class like so:

customException(Exception)

code

and it could then be used by doing the following:

try:

Condition

Except customException()

Your custom exception class will inherit all the attributes of the parent class (Exception) and you can overwrite or modify them for your custom exception class as needed.

**When might you create a class derived from the Exception class?**

While python has dozens of useful built-in exceptions, these just cover generic cases and scenarios that are encountered. If you want to have a more specialized exception that covers a more niche/specific purpose, creating a new class that inherits from the Exception class is a good idea. For example, maybe you have a program that’s used to read in data to a program and non-english text characters (such as Arabic, Cyrillic, Mandarin, etc) will cause problems if they’re loaded into a program. You could create your own exception for this and raise a custom response or message. By default, the exception class doesn’t have anything that covers it specifically.

Creating a child class from the Exceptions class can be a powerful way of creating specific, tailored error messages and exception types (in response to certain conditions) in your code that you can test.

**What is the Markdown language?**

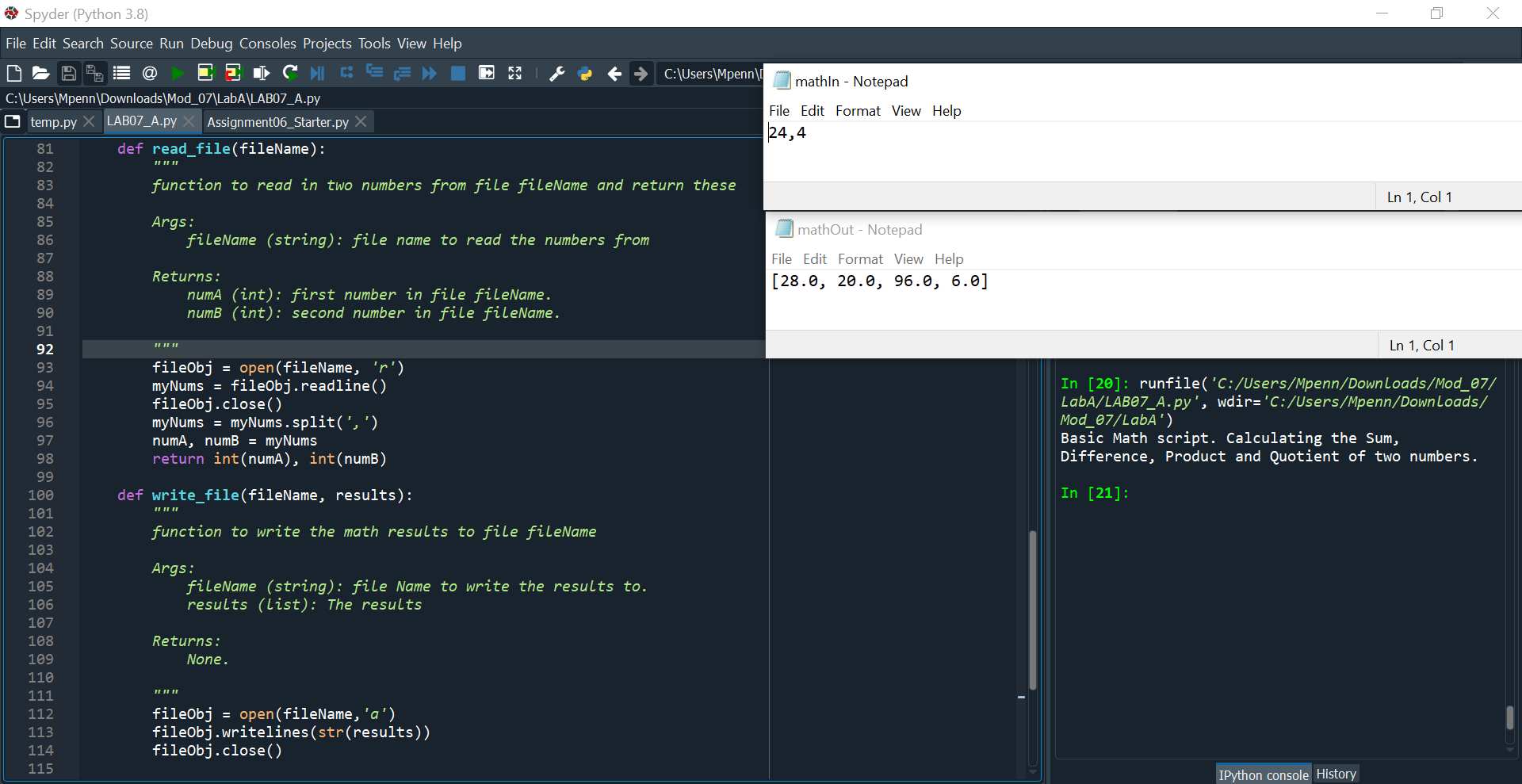
Markdown is a type of formatting applied to text. The most common “marking” language people are familiar with is HTML (Hypertext Markup Language) which uses a series of elements to apply formatting. These are commonly expressed as tags, for example <b>text</b> would be used to make the text between the tags appear as bolded on a page rendered by HTML. There are many more types of formatting that can be applied using HTML, and HTML is not the only Markup language available. Markdown is an alternative way to format webpages that is simpler than HTML. It’s tags are easier to understand and it’s also converted to HTML automatically.

**Labs and Examples Covered**

As noted prior, we were given several labs to cover from module 07, here: https://saravji.github.io/saravjis\_hut/FDN\_Prog/Modules.html (external site)

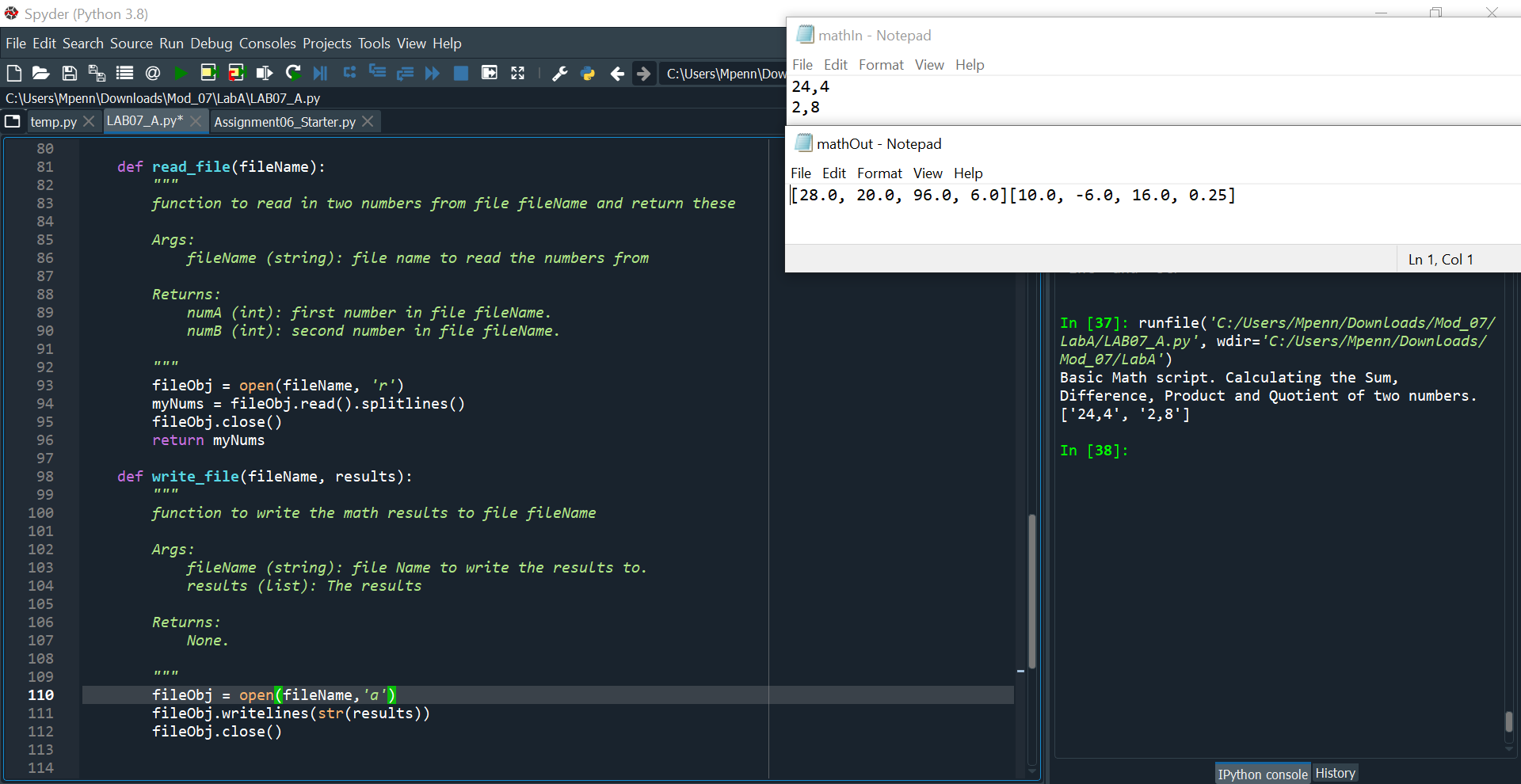
I will discuss in detail these labs below and show output from them.

First, we covered lab A:

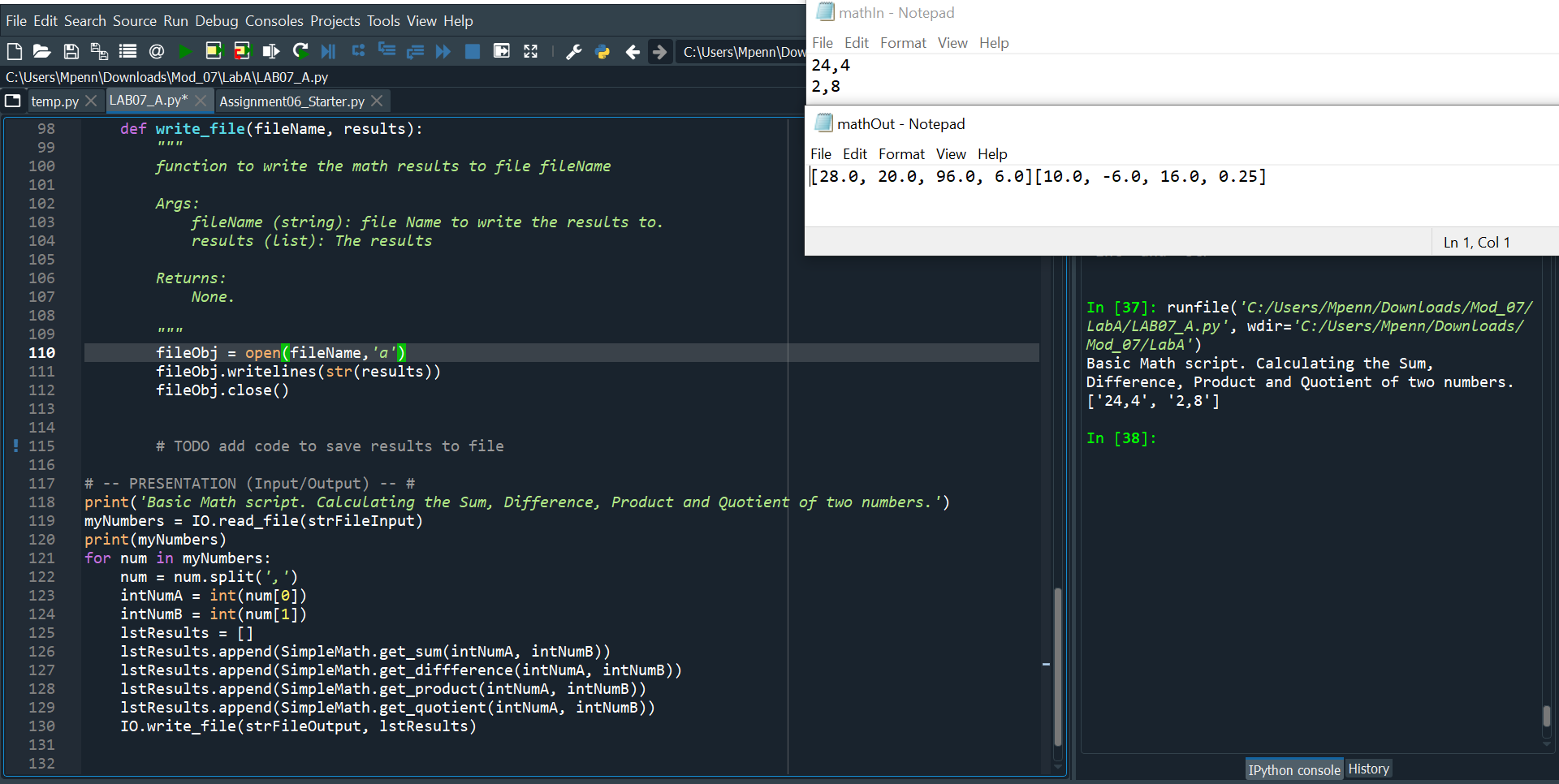


**Fig 01. Reading and writing a single set of values to and from a file**

The first part of this lab had us complete two TODOs: first, we needed to read in two numbers from mathIn.txt to be calculated. This was simple as we used the open() method along with the filename in the variable and the ‘r’ option for read. We then use readline() to read in the single line, close the file, and then split on the comma to get the two numbers that are returned. Next, we needed to work in the write\_file function to write it out to a file after mathematical operations were completed. I just use the ‘a’ (append) option here instead and writelines() to write it to the file before closure.

****

**Fig 02. Updated code to accept multiple lines**

****

**Fig 03. The rest of the updated script**

Next, I modified the file to accept and output multiple lines of arguments. I had issues with the newline character getting stuck on the end of lines coming in, so I used the read.().splitlines() function combination to fix that. This also saves lines, so it was all I needed to get the two sets of values.

I then have a for loop to split each set of numbers at the comma and assign each one via their index number to a variable that then runs it through all the calculation functions and puts the results in a list. This list is written to the file for the number of lines iterated over.

Next, we covered lab B which was a bit challenging:

A picture containing text, screenshot, monitor, computer

Description automatically generated

**Fig 04. Running lab B script with IO and calc options on the command line**

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig 05. The calc/IO logic branches**

One requirement of this lab was taking an argument from the command line: IO or Calc and then doing specific actions based on the chosen command passed via sys.argv. In this case, I use a statement to check for a length of two (confirming an argument was passed) and then use a check on the value stored at index 1, which is where the first of an argument is passed. IO asks for two numbers, calculates them, and then has them written to a binary file (shown below). For the calc branch, we just read in arguments from a file and do the same thing before calling the write\_file method again.

A screenshot of a computer

Description automatically generated with medium confidence

**Fig 05. The read/write file methods that read/save a binary file**

Lastly, I updated the write\_file and read\_file methods to use the pickle.load and pickle.dump methods to read data from binary files and save to them respectively. This took longer than expected as I didn’t know I needed to save out a binary file first before I could read it in.

After completing this lab, I worked on Lab C. This involved taking the code from Lab B and working in error handling where needed.

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig 06. Try/Catch for a ZeroDivisionError. Supplied values manually for demonstration.**

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig 06. Code running as expected. Values manually supplied for demonstration.**

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig 07. Triggering TypeError condition. Values manually supplied for demonstration.**

Text

Description automatically generated

**Fig 08. Triggering TypeError condition. Values manually supplied for demonstration.**

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig 09. Code running as expected**

For this lab, I commented out a lot of the input/output code and tucked sets of manually declared variables above instead of having to run the commands from the command line. I then set up a single try condition and two exception blocks: one to catch the user trying to enter zero as an argument, triggering the “ZeroDivisionError” block and the “TypeError” block to catch exceptions related to the user supplying string data as an argument. I then set at least one value to something that was guaranteed to throw an exception: 0, None, and ‘jello’. The first throws an error when we try and use it to divide 16. The “None” value type and a string of ‘jello’ are also used, triggering the TypeError block. In both cases, an error message is displayed and the program ends via sys.exit() to avoid processing the invalid arguments any further.

Lastly, we worked on the main project for the Module. This involved writing in error handling and pickling data methods into our previously worked on CDInventory project:

This proved to be the hardest lab for me so far, and there are two bugs I’ll note that I wasn’t able to fully weed out. I spent hours trying to fix them, but didn’t have much luck.

First, I updated the read\_file method:

Text

Description automatically generated

**Fig 10. The read file method with updates for pickling and error handling for FileNotFound error class**

This function I was never fully able to get to work and produces a bizarre bug. No matter what content has been written to the file using the functional write\_file method, the program errors out on line 103 when I invoke the pickle.load() method on my file with the output “EOFError: Ran out of input”. Despite opening the file in read mode, the script actually deletes all the content in the file it attempted to load upon failing.

I was unable to find any resources or others experiencing this problem, as I didn’t think that opening the file in ‘rb’ (read) mode could actually affect the content of the file. If you want to replicate this bug, do the following:

1. Create entries in the table using the ‘a’ option.
2. Write these out to a file using the ‘s’ option – you should see your entries there in a binary format.
3. Run the ‘l’ option to load from file, invoking the above code. I disabled the line that calls the method before the main loop, as that was causing the program to crash and continually erase my file contents – but it can still be manually called with ‘l’

This will throw the error, and erase all the content of the .dat file you attempted to load. Despite around six hours of trying, I was unable to resolve this issue. Next, I rewrote the write\_file method and encountered another bug:

Text

Description automatically generated

**Fig 11. The rewritten file\_write method with error handling and pickling**

The catch this time is that the code will only work outside of the function section. I found this out after a few hours of trying to fix it – this exact same code is also in the main script body where the method is normally called. All the function will do is save to an empty file. This indicated to me that the list is somehow not getting passed into the function at the function call, despite the proper handover to the method being present.

A picture containing text, monitor, screenshot, computer

Description automatically generated

**Fig 11. Exact same code in the main body. Saved data shown in the upper right, method code only makes empty file**

Despite the strange bugs noted above, I did have success writing in error handling for the delete entry function:

A picture containing text, monitor, screenshot

Description automatically generated

**Fig 12. Delete function preceded by a try/except block. Triggered the ValueError message by entering a string**

This was pretty simple – we try and cast the input to an integer. If the user enters a non-integer value, the exception is triggered and the break statement is triggered.

Next, I added a try clause to the “add CD” function. One of these entries (the ID) must be an integer in order for the rest of the code to work:

A picture containing text, monitor, screenshot, screen

Description automatically generated

**Fig 13. Try/except block attempting to cast the ID to an int. Prints error message and terminates program if exception occurs.**

In this case, when the valueError exception occurs – I have the program stop. This prevents an entry from being appended with no ID as the code would otherwise accept the other two values and still store it with invalid data. With this, we’ve now included error handling in the areas where data is loaded, saved, or altered by a user.

**Summary**

This module contained some challenging content, namely in relation to pickling and unpickling which ended up being harder in application than I expected. Error handling was also a welcome topic as I’ve had my programs crash before and it gives us a smooth way of handling it. Lastly, the recap on working with regular text files as a good memory jogger and helped reinforce the concept. I spent far more time on this module than the others (namely trying to fix the two bugs noted). Please feel free to leave details on how they could be fixed as I did try quite a few things to no avail and the bugs themselves are strange to me. Your feedback will be greatly appreciated and I can update my code to work as expected; I’m very curious and perplexed by the bugs. The error produced by the read\_file() function will be included below and instructions have been left on how to easily replicate it.

**Appendix**

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig 14. The EOF error generated by pickle.load() – deletes file contents and states the file is empty, regardless of how much content is in the file.**

Text

Description automatically generated

**Fig 14. Adding data in anaconda**

Graphical user interface, text, application

Description automatically generated

**Fig 15. Writing data using anaconda prompt**

Text

Description automatically generated

**Fig 16. Triggering the try/except block by passing an invalid string value for ID**

Text

Description automatically generated

**Fig 17. Triggering the try/except condition in add\_cd by supplying a non-integer value**

Text

Description automatically generated

**Fig 18. The buggy read\_file() function crashing on the pickle.load() line on a file with content in it – destroys file content upon failure.**