Sean McElmurry

2020Mar09

Foundations of Programming, Python

Assignment07

Binary Files and Structured Errors

# Introduction

Over the seventh week of the Foundation of Programming, Python class, we have learned about using binary file types how to use structured errors. To apply the information, we modified our script from the prior week’s assignment. The execution of the script in a terminal window can be seen in Figure 1 below.

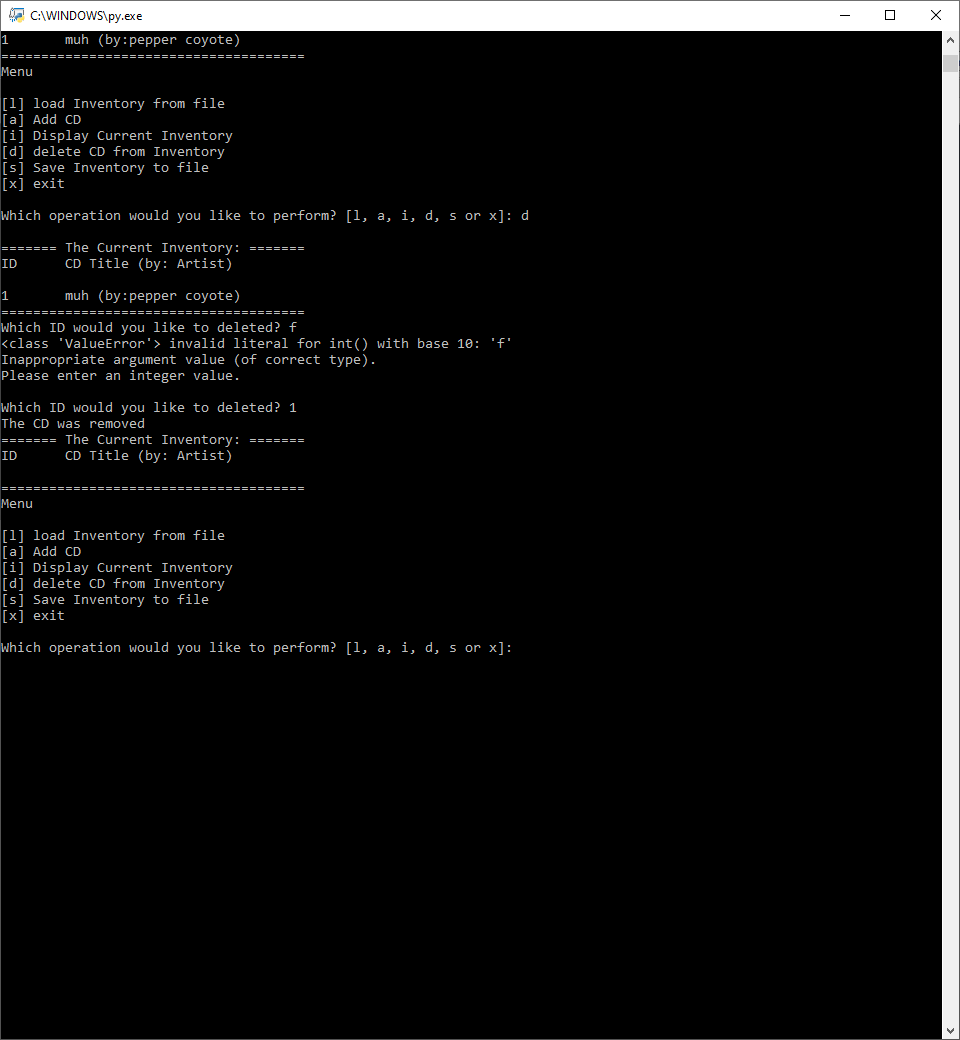
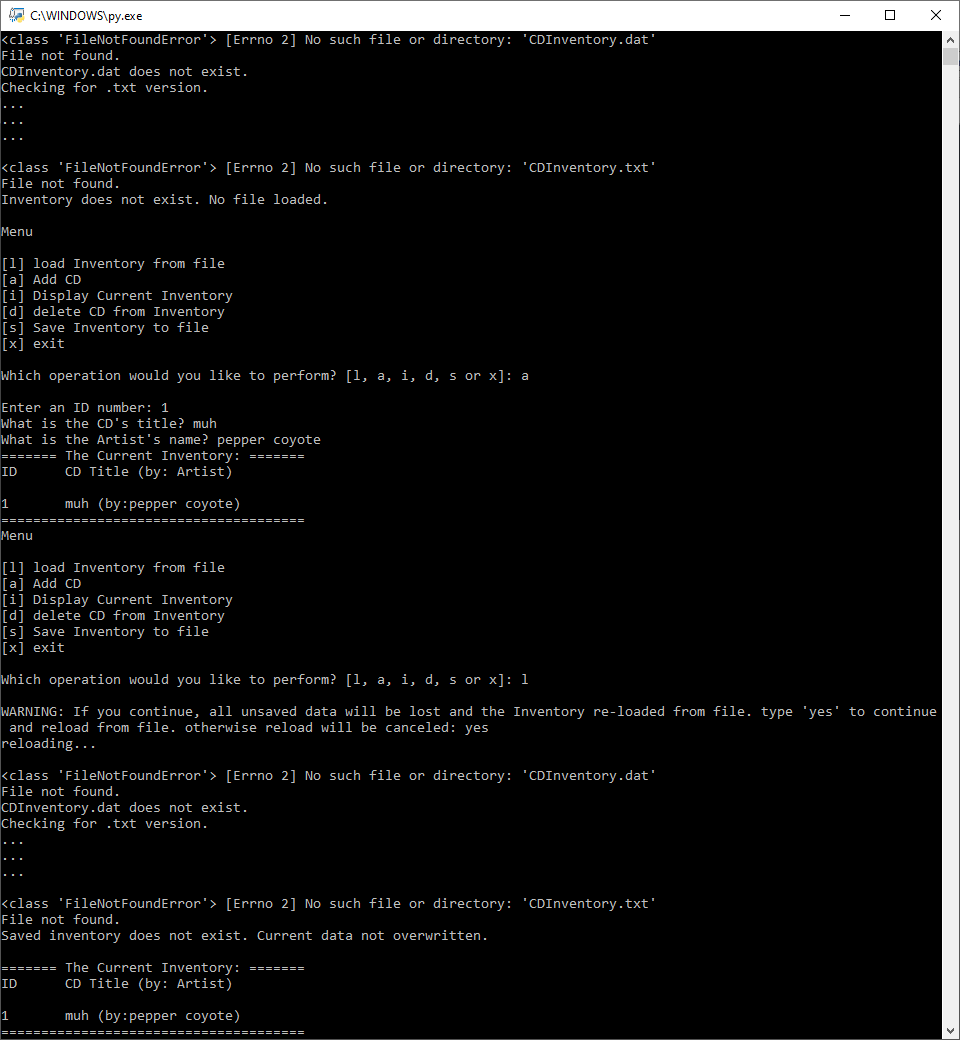


Figure 1- CDInventory.py When Run from a Terminal Window

# Binary Files

Text files are useful when reading plaintext files on different systems, as the format is the same between operating systems, but they cannot store complex data types. For that, it is more convenient to store our information in a binary file. While these types of files are not easily readable to the human eye, when using binary files, we do not need to write functions to convert information to or from these types of files. And in order to store and retrieve from those files, we have two options at our disposal: pickling and shelving. For our current purpose, we only need pickling.

When researching pickling, I did find a site that states "pickle can save and restore class instances transparently" and goes on to explain that this is possible if the class itself is stored in the same module as the data referring to the class type ([*Understanding Python Pickling with Example*](https://www.geeksforgeeks.org/understanding-python-pickling-example/), External Site). Does this mean that virtually anything could be saved using a pickle file? And that we could even save other .py files this way?

# Exceptions

One of the main benefits of using structured error handling is the ability to catch sections of codes where errors could occur, identify and provide feedback on the errors, all without forcing the script to stop the way it would if the program had been run without structured error handling.

The Exception class contains objects that are representative of the errors encountered when running code. Using the class it is possible to obtain information to display to user to help better understand the error such as the type or error, or what item was responsible for triggering the error.

To derive a new class from the Exception class, follow the class title with (Exception). This lets python know which class (Exception) to derive class properties from ([*Python Custom Exceptions*](https://www.programiz.com/python-programming/user-defined-exception), External Site). This does have me wondering if it works for other class types as well? Derived exception classes can also inherit the traits of specific exception as opposed to the general exception class.

A custom exception class is usually created when there is a specific error an author wants to catch. In good practice, these custom errors are not real errors (as those exist in the Exception class already) but are program specific to restrict data within script-specific bounds ([*Writing and Using Custom Exceptions in Python*](http://www.codementor.io/@sheena/how-to-write-python-custom-exceptions-du107ufv9), External Site).

# Markdown Language

While not required while writing the script, we also learned about the Markdown language. The Markdown language is a type of formatting language for plain text. At a first glance, Markdown seems like the Python of HTML, a more intuitive and simple language for achieving the same result as previous languages ([*Basic Syntax*](https://www.markdownguide.org/basic-syntax/), External Site). In our class, we will be using GitHub Flavored Markdown to change our README files in GitHub.

# Writing the Application Script

I think the biggest challenge to this week’s script was learning how to set up exception classes. Once I learned how I wanted to set up error messages in the *read\_file()* function the others came much more quickly. I had almost forgotten to include the update to use binary files, but luckily that information seems more intuitive since it’s a tool that helps reduce the number of lines from our original script. The full script can be seen by following the GitHub link in the appendix; and its execution in the Spyder terminal window can be seen in Figure 2 below.

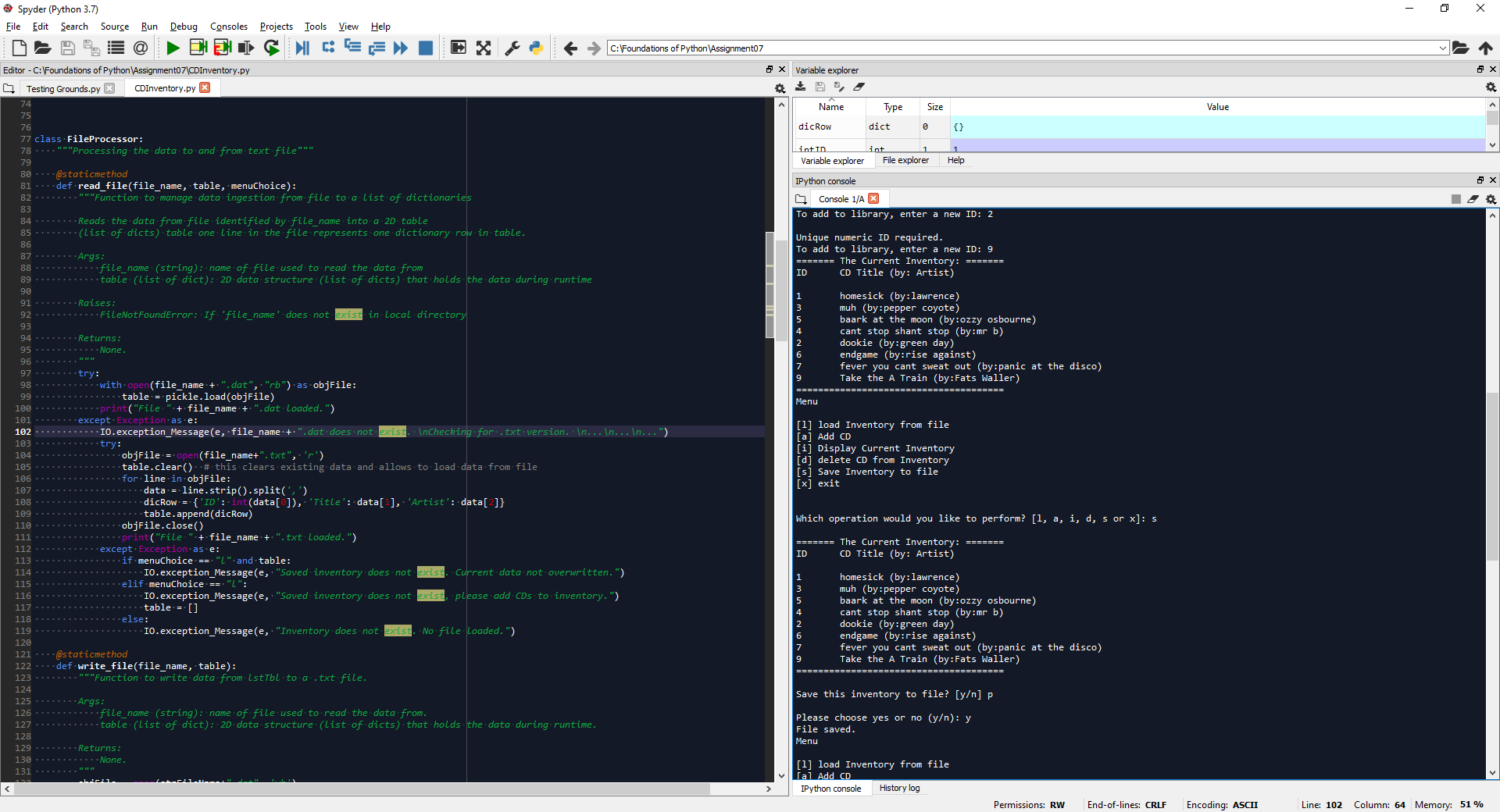
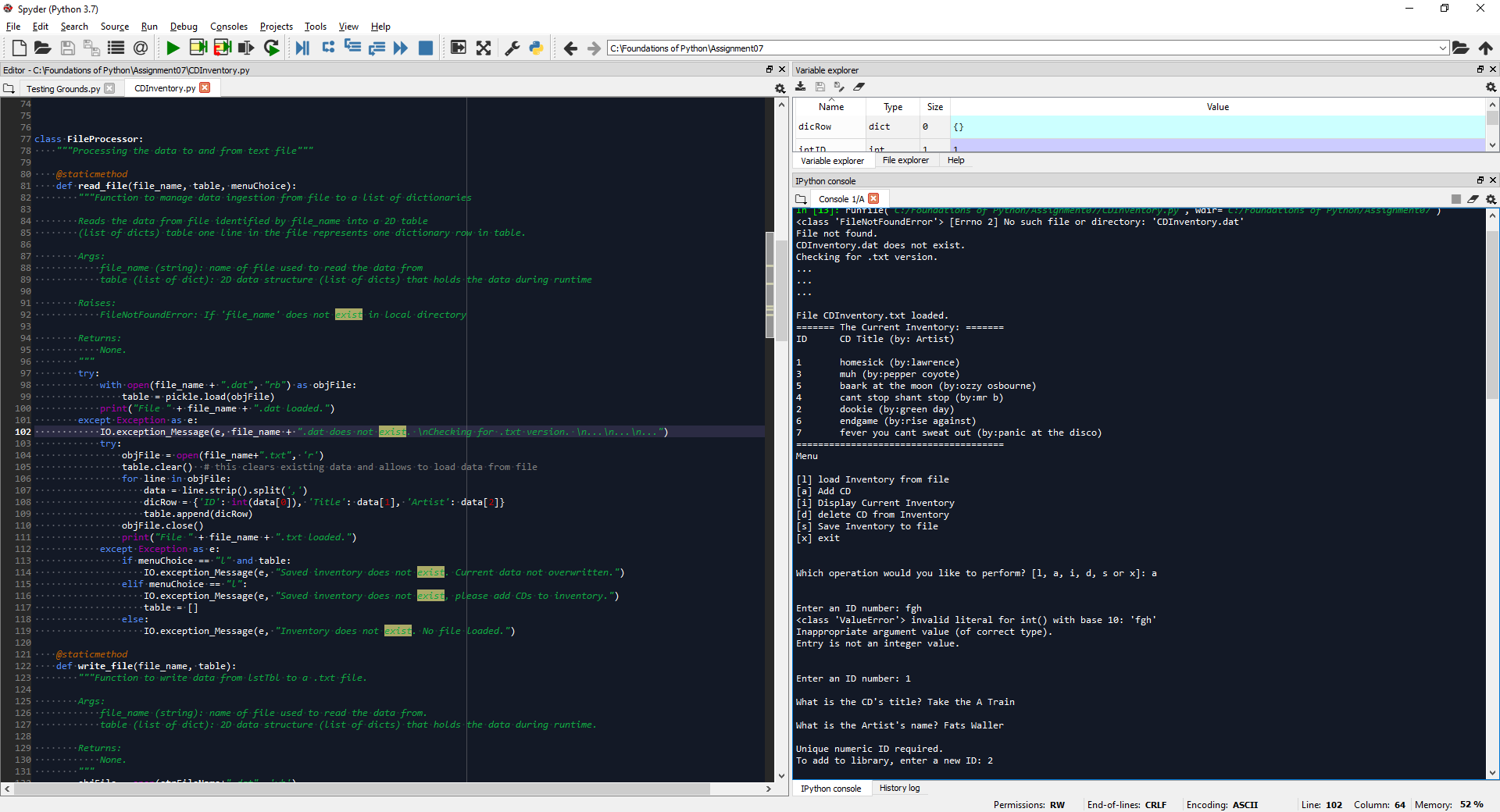


Figure 2- CDInventory.py Run in Spyder

At first I tried the same approach as the previous two assignments and tackle each section of code by the menu choices, but since we’ve added functions the code, addressing the problem in this linear way proved inefficient. Instead, I chose to address the changes by error type: File Access, Type Casting, and User Interaction.

## Exceptions

I think once I learned that Python will generate the exception type for you when an error is triggered, it saved me the most time. Prior to this, I kept trying to define the type of exception for it. However, I still think that triggering errors when setting up the exceptions is extremely helpful as you know what is going wrong, and it helps us set up a custom string to relay to a user so that they know what went wrong and how to avoid it in the future.

I started with the *read\_file()* function and just loading the text file that we worked with the week prior since it was the simplest error to trigger by simply not having a file to load. I decided to relay three pieces of information to the user that I believe help explain the user as Python sees it to the user (exception type, the exception itself, and the exception documentation). Lastly, I added a custom message at the end of the function to explain to the user how to fix it in relation to the script. Once I had this section of code set up, I decided I could use it for all exception in the script and created the *IO.exception\_Message()* function to call whenever an exception arose.

Then I spent way too long trying to create different custom messages for that same error based on if information had been written into a list when trying to load an empty file.

I decided the *write\_file()* function did not require any exception when it came to file access under the assumption that the data to be written would be in the correct format (because of other error catching structures) before the function is called.

After addressing file access errors, I moved on to type casting errors. Luckily for me, the feedback I received from last week’s assignment helped me speed up this process greatly. And the *exception\_Message()* function allowed me to shorten the new *value\_Errors()* (previously *int\_check()*) function to just six lines of code as seen below.

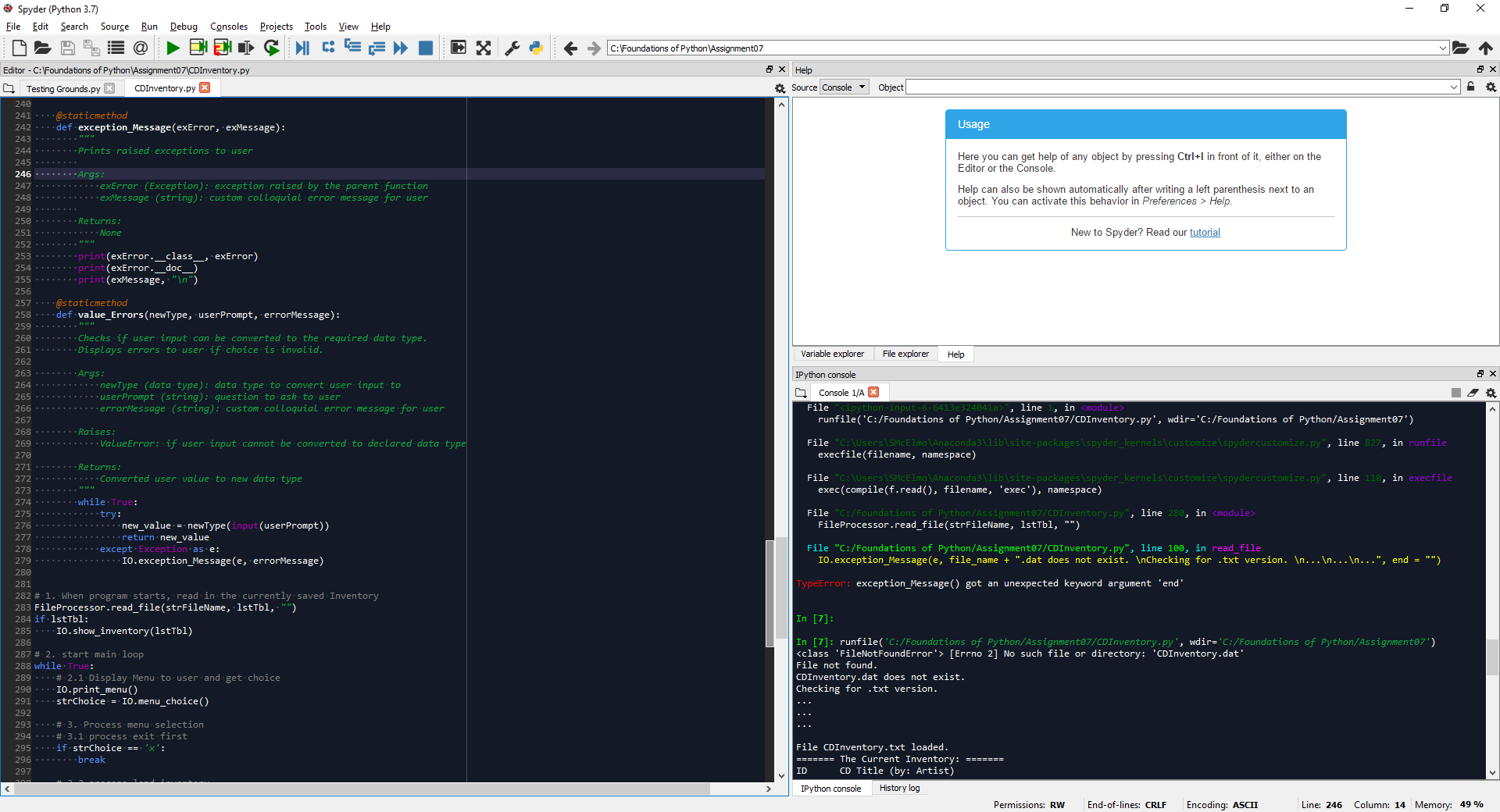


Figure 3- IO.value\_Errors(newType, userPrompt, errorMessage)

That function in tandem with *exception\_Message()* addressed all of the type casting errors in the script that I felt we needed to be concerned with. In ­*read\_file()* and *write\_file()* I felt that those type casts did not need to be addressed under the assumption that the files being loaded would already be in the correct format, or would be ready to be exported in the right format respectively.

Finally, addressing user interactions, most interactions that needed to be addressed with the user involved type casting, so after identifying where users could enter values, I looked at the types of errors at those spots. Almost all errors had already been addressed by the edits made above. The only real area of concern left was the fact that the yes/no questions could take values other than yes or no, so I made a new function (*IO.yes\_No()*) to call on whenever a simple question like that is asked.

## Binary Files

I was about halfway through writing this document when I saw the last line instructing us to update the code to interact with binary files. Luckily, working with binary files came much more easily than exceptions. Not only is accessing binary files similar to accessing text files, but the commands in Python are much simpler because we don’t need to worry about data types.

I wasn’t sure if we were supposed to worry about text files still so I added a section to *read\_file()* to attempt to read a .dat file first, then a .txt file as a secondary measure just in cast the binary file doesn’t exist, and we attempted to use last week’s data. At first, I was thinking about making a try/except statement with multiple exceptions, thinking it would be what I needed. But it turns out I was able to use a nested try/except instead.

# Summary

Now that I know the information I prefer to show to a user when triggering exceptions, setting up structured error handling in the future seems like it will be much simpler. I think one of the most useful tips I picked up was how to use a *while True:* loop with no Boolean exits. It’s a nifty way to ensure the user is forced to enter the correct data eventually before returning the data.

# Appendix

## Sources

Cone, Matt. “Basic Syntax.” *Markdown Guide*, www.markdownguide.org/basic-syntax/. Accessed 2020Mar08.

“Python Custom Exceptions.” *Programiz*, www.programiz.com/python-programming/user-defined-exception. Accessed 2020Mar08.

Sheena. “Writing and Using Custom Exceptions in Python.” *Codementor*, 14 Mar. 2017, www.codementor.io/@sheena/how-to-write-python-custom-exceptions-du107ufv9. Accessed 2020Mar08.

“Understanding Python Pickling with Example.” *GeeksforGeeks*, 13 Nov. 2018, www.geeksforgeeks.org/understanding-python-pickling-example/. Accessed 2020Mar07.

## GitHub Repository

<https://github.com/SMcElmo/Assignment_07>