Alok Malavia

2022 Mar-13

Foundations of Programming: Python

Assignment\_07

Python – Working with Text and Binary Files. Structured Error Handling

# Introduction

In this document we cover the topics of working with Text Files, Binary Files and also taking a look at the Structured Error Handling. In previous modules, we covered working with text files, where we wrote, appended, and read from files, but in the Module this week, we further explored the concepts by looking at writing, reading, and appending to the text files by using them in functions. Specifically looking at reading only one line or multiple lines from a file. Finally, we look at the concept of structured error handling so that we can gracefully continue our program based on the expected error or close out of the program, if necessary.

# Reading, Writing/Saving, Appending Data

Until now in our class, we have worked with saving data files in plain text files. The benefit of these files is that they are easily human readable. In this module, we specifically focused on reading the data saved in files, but also the ability to save\_data or write the data to file given the relevant scenarios. We looked at examples in the module where we could define a function where we assigned values to global variables such as strData, and strFileName. The strData variable could be any string that you wanted to be saved to a file, and the strFileName variable would be the name of the file to which you want to save data. We then were able to define a function where we provided the data, and file\_name arguments, which correspond to the strData and StrFileName respectively. What this does is that it saves ‘data’ to the ‘file\_name’. In this case, since we are saving the data, we don’t need to return any data, therefore our return argument was none. Once the data was saved, we could read back the data from the defined txt file in strFileName, and this was done via a read\_data function. This function allowed us to read data from the file which we defined for the variable strFileName and was able to provide the relevant output. These two basic functions allowed us to write data and read data from the file that we created. In our example, when we typed in the data which was also reflected in the file, but also, we were able to read from the same file back. This showed us the power of functions, when it comes to working with files, both writing and reading data from them. If we needed to add any additional data to the file, this could be accomplished by using a append\_data function, and in our calling of the function, we would simply identify the additional data being added, followed by the strFileName variable, so that we could call the file to which the data was being appended.

## The Reading Data Options

We also covered various options to read data from files. We learned how to use the readline() and readlines() functions to specifically pick out certain data that we wanted to be displayed. Additionally, if we wanted a specific line to be displayed, this could also be accomplished by using the redline function. One of the important concepts we learned out of the readline and readlines function is the concept of a ‘cursor’. On each call of the readline() function, the next line(one line) of data gets returned. As long as the access to the file stays open, the location of which line to read next is then memorized, and this concept in programming is generally called a cursor. A caveat to the cursor rule is that if you close the file that you are working with, it will reset the ‘cursor’ to line 1, which means that your readline function may return a line that you were not expecting, i.e. it will read you line 1 again. Finally, when you are using the readline or readlines functions, you can use them as a loop to iterate over the data in the file, or you they can be used as a ‘with-as’ option as well.

# Using and working with Binary Files.

So far when we are saving data to files, we typically have been using text file. In text files, we are using plain text or strings to store our data, so they are easily readable to the end user. However, in programming that is not typically the way in which information is stored in the memory. It is not always necessary to change the data in a readable format, the easiest solution would be to save the information as it is in memory. In essence, this is the Binary code. This is especially practical to save states of your application or settings the user made. Saving binary information in Python is called pickling. In python we have a module called pickle and it takes information associated with an object and serializes or (de-serializes) it in a way that it can easily be stored or loaded as binary information. Although we covered a few examples in class, I was able to go online and find some good materials that talk about Pickling data. I found that that [this](https://www.geeksforgeeks.org/understanding-python-pickling-example/) example on the geeksforgeeks site really provided a simple example and was really good at explaining the concept of Pickling data through a list of dictionaries. Although most explanations of Pickling on the web were good, I thought that [this](https://www.synopsys.com/blogs/software-security/python-pickling/) page wasn’t great at explaining the overall concept of Pickling in Python. Overall, I think pickling is still a tough concept to grasp my head around, but hoping with some practice, I can get better at using the module correctly.

# Structured Error Handling

Up until this module, we have been running our programs and if there is an error while running our programs, Python will scream at us telling us there is an issue. Although, the error provides us information of why the program crashed, it is rather abrupt and provide information that is hard to understand at times.

Text

Description automatically generated

Figure - Generic Error from Python without Error Handling

To address the issue above, we discussed the topic of the ‘try-except’ construct in the last module. This construct in our code allows us to protect the parts of our program in a structured way, where an error is expected as result of either human error or the program receiving a value other than what the program is looking for. It is also commonly referred to as ‘trapping’ the statements of our code, so that we can customize and handle the errors that come our way. This overall concept is referred to as structured error handling as we are capturing the exception object and extracting details about the error. One of the links I found that talks about a basic example of error handling is shown [here](https://docs.python.org/3/tutorial/errors.html). However, as our codes get more complex and have additional data, we can look to some more complex ways of doing error handling, and I found such examples [here](https://stackoverflow.com/questions/5627425/what-is-a-good-way-to-handle-exceptions-when-trying-to-read-a-file-in-python). I would say that the first link was a simpler way of looking at handling errors and using built-in [Exceptions List](https://docs.python.org/3/library/exceptions.html) , we can come up with efficient ways to handle our errors. We also covered the topic of catching very specific errors, and they can be handled by creating Custom Exception Classes, which allow you to customize how you want the errors to be handled based on your specific needs.

# Assignment\_07 Examples of Pickling and Structured Error Handling

The goal of Assignment07 was to take our CDInventory.py file from Assignment06 and add structured handling to the areas with user interaction, type casting (string to int) or file access operations. Initially, error handling seemed to be a challenge, but overall looking at it in action, provided clarity on how to make it work. Additionally, I was able to incorporate error handling into the script Assignment06.py that we modified as part of this assignment. Finally, I was also able to use and store data to our file in Binary form. When we run the modified script, we give the user an option to pick one letter of the six letters provided. I added error handling so that the script would keep looping until the user selects one of the six letters provided.

Text

Description automatically generated

Figure - Structured Error Handling - 1 - User Interaction

The script than continues into it’s next portion which could be to add information to the CD Inventory, because the first function called is the read\_file function, and the first operation in the function is to clear the existing table that might exist. Therefore, we typically proceed with adding data. Here is another instance, were I added error handling to address user interaction that specifically deals with a string vs. an integer input. As shown below, if the user did not enter an integer, the program would not proceed to the next step.

Text

Description automatically generated

Figure - Structured Error Handling - 2 - User Input - Int vs String

We continued working through our script and based on the input of the user, we incorporated error handling to address An additional area where we worked with structured error handling was when we are working with files and reading and writing to files. If there is no file to work with, In this assignment, I added error handling to provide the user with a message that there is no file and to add a file before proceeding. However, in this portion of the assignment, I was struggling with figuring out how to exit out of the program until the user was able to load a correct file. See screenshot below:

Text

Description automatically generated

Figure 4 - Structured Error Handling - 3 - File Interaction

As we continued working through the program, we were able to add data, display data, save data, and one of the interesting wrinkles of this assignment was to store the permanent data in a binary format. This was done so by modifying the code in the read\_file and write\_file functions by using the pickling module. We imported the pickling module by typing the ‘import pickle’ command at the very beginning of the script, and modifying our File Processing class and functions, by using the pickle function. Please see the figures below:

Text

Description automatically generated

Figure 5 - Importing the Pickle program in the Module

Text

Description automatically generated

Figure 6 - Pickling Data - pickle.load and pickle.dump

# Example of the Script Working in Spyder and a Terminal Window

Below are screenshots of the script being executed in the Spyder console window:

Text

Description automatically generated

Figure 8 - Script execution in Spyder console

Text

Description automatically generated

Figure 9 - Script Execution Spyder - Cont'd

Text

Description automatically generated

Figure 10 - Script Execution - Spyder - Cont'd

Text

Description automatically generated

Figure 11 - Script Execution - Spyder - Cont'd

Script Execution in the Terminal Window

Text

Description automatically generated

*Figure 124 - Script execution in Terminal Window*

Text

Description automatically generated

Figure 53 - Script execution in Terminal Window-Cont’d

Text

Description automatically generated

Figure 14 - Script execution in Terminal Window-Cont’d

Text

Description automatically generated

Figure 15 - Script execution in Terminal Window-Cont’d

# Summary

This document covered the steps taken to add code to a starter script which was provided to us. The goal of the assignment was to modify the starter script to use functions to refactor the code and this time around also add in Structured Error Handling and store data in file in binary. We were able to invoke error handling at the relevant parts of the scripts as specified. We also were able to see that the data saved to files was no longer in .txt format, but rather in binary. The code for this program, including this knowledge document can be found on my github repository.