Arielle Waller

August 26, 2020

Foundations of Programming (Python)

Assignment07

Assignment07

# Introduction

For this week’s assignment, we were to use the Internet to find examples demonstrating how to use exception handling and pickling in Python. Then, we were tasked to explain why we found those examples good or bad. Finally, we had to modify the script from Assignment06 by:

1. Changing the permanent data store from a plain text file to a binary file
2. Including structured error handling in parts of the script that involved end-user interaction, type casting, or accessing files

# Changing the Permanent Data Store

I thought that changing the data store would be the most straightforward part of the assignment, so I started with this task first. First, I changed the string assigned to *strFileName* from ‘CDInventory.txt’ to ‘CDInventory.dat’ since the data storage file would now be a binary file.



Figure 1 - Changing string filename from .txt ending to .dat.

Next, I moved to *FileProcessor* since this class contained the static methods that interacted with my file object. In the method *write\_file*, the code from last week iterated through each dictionary row in the 2D table, created a list of the values for each dictionary row, and wrote each row to the file with the row values separated by a comma and rows separated by a new line.

A screenshot of a cell phone

Description automatically generated

Figure 2 - Original write\_method from last week.

I changed the mode for the *open()* function to *‘wb’* to open a binary file for writing since we were now working with a binary file.



Figure 3 - Opening a binary file object for writing.

I used *pickle.dump()* with *table* and *objFile* as arguments to pickle the 2D table and write it as one object to the file.



Figure 4 - Pickling the 2D table and writing it to the binary file.

As I did this, I realized I needed to first import the *pickle* module and added a line of code at the top of my script to do this.



Figure 5 - Importing the pickle module.

In the method *read\_file*, I had a *try­-except* block that attempted to open a text file in read mode and otherwise printed a message and created a new file. In the *try* block, I changed the file access mode for the *open()* function from ‘*r’* to *‘rb*’ to open the binary file in read mode.



Figure 6 - Opening a binary file for reading.

In the *except* block, I changed the file access mode for the *open()* function from *‘w’* to *‘wb’* to open/create a binary file for writing.



Figure 7 - Opening/creating a binary file for writing.

In last week’s assignment, if the file existed and could thus be opened for reading, my code looped through each line in the file object. Each line was stripped and split into a list based on commas. A dictionary was created for each line by using the strings *‘ID’, ‘Title’,* and *‘Artist’* as keys and assigning them the values at specified indices in the list. Finally, this dictionary was appended to the table (the list of dictionaries).

A picture containing drawing

Description automatically generated

Figure 8 - Original code for reading from a file from last week.

Because my data this week would be pickled, I could cut down on the amount of code required to load my table from the file. I deleted everything in the *for* loop. To unpickle the list of dictionaries stored in the file, I usedthe *load()* function from the *pickle* module with my file object as the argument. Going back to the I assigned the value returned from using *pickle.load()* on my file object back to *table*.



Figure 9 - Unpickling the 2D table and assigning it back to table.

Then, I closed the file.



Figure 10 - Closing the file.

At this point, I stopped to test the code. When I tried to load the inventory from a file, I got an EOFError.



Figure 11 - End of file error.

Through some [research](https://stackoverflow.com/questions/24791987/why-do-i-get-pickle-eoferror-ran-out-of-input-reading-an-empty-file#:~:text=EOFError%20is%20simply%20raised%2C%20because,just%20meant%20End%20of%20File%20..&text=It%20is%20very%20likely%20that,re%20copying%20and%20pasting%20code.&text=You%20can%20catch%20that%20exception%20and%20return%20whatever%20you%20want%20from%20there.)[[1]](#footnote-1) (external site) and experimentation with a *print* statement, I found that this type of error occurred when a binary file existed but contained no data. I added an *else* block to my *try-except* block of code. Here, I nested another *try-except-else* block. I moved my code for unpickling the binary file from the outer *try* block to this inner *try* block.

A close up of a logo

Description automatically generated

Figure 12 - Attempt to unpickle the binary file.

In the *except* block in the inner *try-except-else* block, I used *EOFError* as the condition. If this exception was raised, a statement would display and inform the user that the file was empty. Data needed to be written to the file first.



Figure 13 - Display an error to the user if the file is empty.

If this exception was not raised, the *else* statement in the inner *try-except-else* block would execute and the table would be printed.



Figure 14 - Print the table if no exceptions are raised.

In the *else* statement in the outer *try-except-else* block, I closed the file and returned the table. Previously, I was closing the file in the *try* statement in the outer *try-except-else* block. At this point, my returned table was not assigned to anything in the main code. In class, I learned it really did not really have much of a use then since it was not assigned to anything.

A picture containing drawing

Description automatically generated

Figure 15 - Close the file object and return the table.

I tested the code again. I did not receive any error messages, but the table was empty when I tried to load it after saving data to the file. With a *print* statement, I found that the data was loading successfully, but it was not displaying when *IO.show\_inventory()* was called on *lstTbl*. Through some experimentation, I found that I needed to assign the return value from the method *read\_file* back to *lstTbl*. This was confusing for me because I thought that since a list is a reference type, *lstTbl* was being changed inside of my function already. However, it was successful.



Figure 16 - Assign the return value of the method back to lstTbl.

With this taken care of, I continued to adding structured error handling.

# Structured Error Handing

I had already included some structured error handling for file access in last week’s assignment. In modifying the permanent data store to use binary data, I inadvertently ended up adding more error handling for file access. I did not find any more errors to account for with file access, so I moved to add some for user interactions.

In the class *IO*, I did not find any possible errors that could be generated when calling the method *print\_menu*. With *menu\_choice*, the *while* loop seemed to take care of the user potentially not choosing a valid option. *Show\_inventory* involved output, so that was not a concern. With *cd\_information*, I saw the potential for a *ValueError* since the user’s input for ID was cast to an integer.

I started with a *try-except* block. I wanted to try to cast *strID* to an integer, so I put *int(strID)* inside of my *try* block. In my *except* block, I used *ValueError* as the condition. If there was a value error, a message would display to inform the user that they had not entered an integer. Instead of being brought back to the menu, I wanted the user to continue to be prompted for an ID until an integer had been entered.

I created a variable *IDcheck* and assigned it to *int(strID)* in the *try* block. I nested the *try-except* block inside of a *while* loop with *IDcheck* being equal to *None* as the condition. Above the *while* loop, I initialized *IDcheck* by assigning it to *None.* Thus, the loop would run as long as *IDcheck* was equal to *None. IDcheck* would equal *None* until the user had entered a valid integer. In testing out this block of code, my program entered into an infinite loop when I did not enter a valid integer. I realized I needed to prompt the user for the ID inside of the *while* loop. Otherwise, *IDcheck* would continue to be *None* with each iteration of the loop.

A picture containing drawing

Description automatically generated

Figure 17 - Check to ensure user's input is an integer.

Moving down into the main program, I saw the *else* statement in the main loop took care of users not entering any valid input. The only remaining place I noticed where error handling could be added was in asking the user which ID they wanted to delete. This input was cast to an integer and had the potential for a value error. I copied the code block that I had used for handling the error in the static method *cd\_information*, recognizing that this was probably the point where I needed to make another function.

I modified the variables so as not to confuse myself with the local variables I had used in the *cd\_information* method.

A picture containing clock, drawing

Description automatically generated

Figure 18 - Check to ensure user's input is an integer.

# Running the Script

When I did not see where else to add error handling, I attempted to run the script. I found that I needed to change the argument for the call to *delete\_inventory* from *intIDDel* to *intIDDelcheck*. I also removed the *else* statement in the inner *try-except-else* block in the *read\_file* method since the extra print statement was not necessary. The program consequently ran successfully in both Spyder and the terminal.

A screenshot of a cell phone

Description automatically generated

Figure 19 – Running CDInventory.py in Spyder.

A screenshot of a cell phone

Description automatically generated

Figure 20 - Running CD\_Inventory.py in the terminal.

# Summary

I really enjoyed this week’s assignment since pickling seemed to minimize the amount of code necessary to store and read objects like lists and dictionaries. I also loved learning more about error handling because I had really wanted to know more about ways to prevent my program from terminating. I am still confused about why my unpickled dictionary has to be re-assigned to my *lstTbl* variable.

Sites that I thought explained exception handling well are [python.org](https://docs.python.org/3/tutorial/errors.html)[[2]](#footnote-2). I also liked [python.org](https://docs.python.org/3/library/pickle.html)[[3]](#footnote-3) for pickling. There’s a lot of information on these pages, but I was able to find what I needed with all the details necessary. I also liked [geekforgeeks](https://www.geeksforgeeks.org/understanding-python-pickling-example/)[[4]](#footnote-4) as it was concise and provided concrete examples for pickling. I liked [tutorialspoint.com](https://www.tutorialspoint.com/python/python_exceptions.htm) [[5]](#footnote-5)for the same reasons. I thought the textbook was also particularly helpful this week.

1. Retrieved 2020-August-25 [↑](#footnote-ref-1)
2. Retrieved 2020-August-26 [↑](#footnote-ref-2)
3. Retrieved 2020-August-26 [↑](#footnote-ref-3)
4. Retrieved 2020-August-26 [↑](#footnote-ref-4)
5. Retrieved 2020-August-26 [↑](#footnote-ref-5)