

## Pickling and Error-Handling Script:

### 1.1 Introduction:

This paper describes my script which addresses pickling and error-handling. Using the Separation of Concerns principles, I divided my script into one section for pickling and one for error-handling and pickling together. These sections includes subsections for declaring data, processing data, and presenting or interacting with the user to obtain new data.

### 2.1 Pickling Article:

The article I used to learn about pickling comes from Snyk Blog titled “The ultimate guide to Python pickling” found at <https://snyk.io/blog/guide-to-python-pickle/> [External Link]. This article was helpful because it described the data types the pickle module accepts. It also clearly described the differences between serialization and deserialization, as other articles I had looked at did not address what these actions were and how they were important and relevant to pickling.

### 2.2 Declaring Variables & Pickling Data:

For this section, I defined two variables: a binary file that would be used to store existing and new data, and a list variable that stored the data I wanted to pickle. Knowing that pickling can store and load complex sets of data, I created a list containing three elements. These variables can be seen in **Figure 1**. In **Figure 1**, my list is a variable which stores different book titles, which are represented as strings:

```
11  # Data: Pickling #  
12  pickle_file = "datafile.dat" # a binary file that will be called upon and stored data to  
13  book_titles = ["The Fifth Season", "Tehanu", "La Gesta dels Estels"] # loads in a list of 3 elements
```

*Figure 1 — The binary file that will store data and the existing list that will be used for this section of code*

For consistency within my script, I used the with statement and open() function throughout my section on pickling. This allowed me to review the parameters the function took, how the exact syntax worked, and allowed me to practice in understanding my own code as I wrote it. Therefore, each processing section for pickling in the following figures will look slightly similar, although object file names will look different.

After declaring my variables, I imported the pickle module. Since my data file was already declared, I could call and store it as an object file using the with statement and an open() function. Since this was the first time I would call the binary file, I set the object file's name to 'file' (**Figure 2**).

**Figure 2 — Importing the pickle module and using with and open() to pickle the existing list from Figure 1**

To remind myself of how pickling works, I added comments to each section. Specifically, I added that the `with` statement would close the function once it finishes working with the file. This allows me to not have to add a `file.close()` statement after the data is dumped into the binary file.

Running this section of the script and opening the .dat file showed that the book titles from the list had been successfully dumped into the binary file. As mentioned by Professor Root and other articles I had read on pickling, **Figure 3** shows that the book titles are still in a readable format, but that the rest of the file contains unreadable characters:



The next step was to unpickle the stored data back into the script and display it to the user. I used a similar with-open() statement as I did in section 2.2, but this time renamed the file object that would open the pickled data as load\_file (**Figure 4**).

```

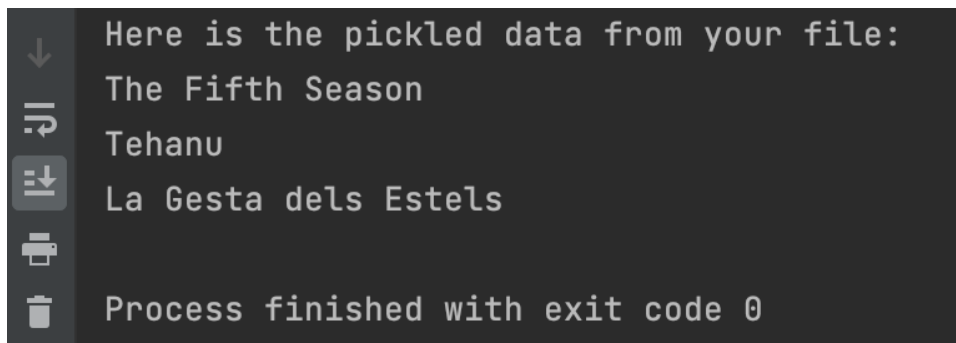
20 # Processing: Unpickle the file (load pickled data in) #
21 with open(pickle_file, "rb") as load_file:
22     existing_data = pickle.load(load_file) # data in binary file is stored in this variable
23     print("Here is the pickled data from your file: ")
24     for item in existing_data: # loops through data and prints it out in a readable format
25         print(item)

```

**Figure 4** — Code that loads in and unpickles data from binary file

In the with-open line, I again called the .dat file, but this time set the mode to “rb”. In this mode, “rb” will read the pickled data from the binary file and assign it to the file object load\_file. I also created a local variable, existing\_data, that would load the pickled data from the file object. By creating a new local variable, I can use it and call the variable should I want to display the data to the user.

As we saw in **Figure 3**, the binary file does not present the most user-friendly or readable format for the pickled data to be understood by the user using my script. To remedy this, I used a for-loop to loop through the variable existing\_data. Since existing\_data has access to the binary file, it can easily loop through the data and print out each element from the original list. Before the for-loop, I included a print statement that reminded the user what this data is and where it comes from. **Figure 5** shows the results of running the code chunk from **Figure 4**:



```

↓ Here is the pickled data from your file:
↻ The Fifth Season
↻ Tehanu
↻ La Gesta dels Estels
🖨
🗑 Process finished with exit code 0

```

**Figure 5** — The unpickled and printed out data that had been stored in the binary file

## 2.4 Taking in and Pickling New Data:

To ensure that I understood how pickling worked, I wanted to try adding in some user-input data to the script. I created a new section called I/O: Pickling New Data that would take in a user’s new book title, append that title to the existing list, and pickle that new list to the binary file.

**Figure 6** shows this process:

```

27 # # I/O: Pickling New Data #
28 print('\n')_# Add new line so that existing data
29     #_and_user_input_prompt_doesn't_look_too_cluttered
30
31 new_book = str(input("Write the title of a book you'd like to read: "))
32 book_titles.append(new_book)_# appends new data to existing list
33
34 with open(pickle_file, "wb") as reopen_file:
35     new_data = pickle.dump(book_titles, reopen_file)_# writes the new data to the file
36
37 print("Books To Read (New!):", '\n')
38 for item in book_titles:_# prints out updated list
39     print(item)

```

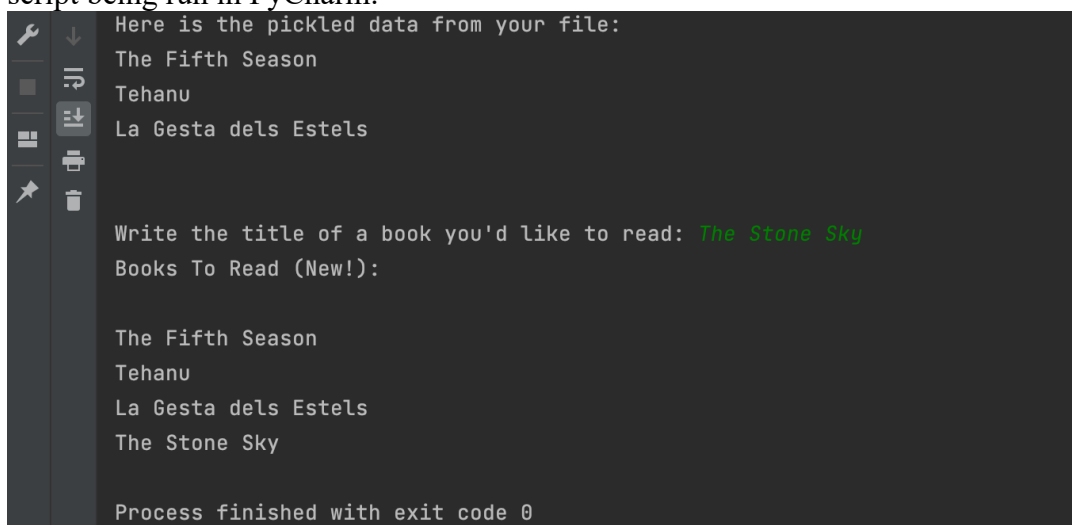
*Figure 6 — Taking in user-input data and pickling it to the file*

In this section, I first added an additional new line so that when this script is run in PyCharm or a Command Shell, the existing list that is loaded in and presented to the user is separated from the user-input prompt.

I created a new variable, `new_book`, which took a string input from the user and asked them to input a title of a book that they'd like to read. Using the `.append()` function, I then had that title be appended to the existing `book_titles` list.

I used similar code from **Figure 2** to write the updated list to the binary file. Calling on the binary file and using “wb” again to write the information to the binary file, I set the file-object as `reopen_file`. Since the data in `book_titles` had been updated through the `.append()` function, the data that is dumped into the binary file includes the new user data that was captured from the input prompt.

Similar to **Figure 4**, I used another for-loop that would loop through the updated list and print out each book title for the user. **Figure 7** and **Figure 8** show the entire pickling section of this script being run in PyCharm:



```

Here is the pickled data from your file:
The Fifth Season
Tehanu
La Gesta dels Estels

Write the title of a book you'd like to read: The Stone Sky
Books To Read (New!):

The Fifth Season
Tehanu
La Gesta dels Estels
The Stone Sky

Process finished with exit code 0

```

*Figure 7 — The pickling section running in PyCharm, showing the first list, the user's input, and the new list with updated data*

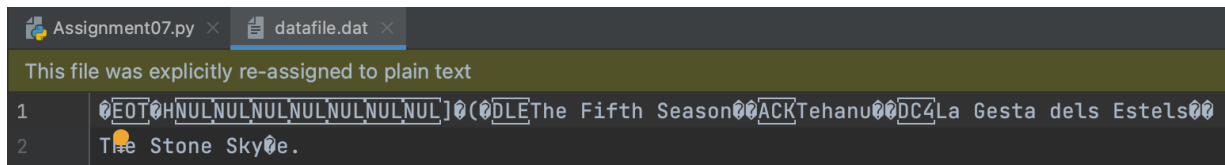


Figure 8 — The binary file opened in PyCharm, showing the updated list

### 3.1 Structured Error Handling Article:

The article I chose was the “Python Exceptions” article from <https://www.javatpoint.com/python-exception-handling> [External Link]. I liked this article because it explained the differences between syntax errors and exceptions errors. Most of my issues while writing the previous assignments had been syntax errors and it was nice to have an explanation about how these two errors are different in Python.

I also found it helpful that within a try-except block, else can be included to catch other errors or perform other functions that the try and except lines do not. It reminded me of the elif-statements that we use when writing if-statements.

### 3.2 Combining Pickling and Structured Error Handling:

On the first pass of my script, I had included a separate section for error handling and one for combining both pickling and structured error handling. I felt that the code was a bit repetitive for both these sections, so I ended up grouping both code chunks into one section.

Structured Error handling involves using Try-Except blocks to catch errors and present those errors to the user in a user-friendly and readable way in case they aren’t familiar with the error names the Python virtual environment throws. The try portion of the block runs a section of code includes a possible way of the code being tested to run and work, while the except block is meant to catch either specific or general errors that the programmer wants to test and improve.

In **Figure 9**, I included a while-loop and two initial values that prompted the user to input two different numbers. Once it had stored those numbers, the try loop is executed, attempting to divide value1 over value2. If that division is possible, then the quotient is stored into a new variable, new\_quotient.

I then created two with-open statements similar to the ones discussed in the pickling section. For the first statement, I converted the quotient value into an integer and had it dumped into the existing binary file. I also included a print statement that told the user their data had been saved, as the .dat file would be closed as soon as the open() function ends.

```

41 # Combining Pickling and Structure Error Handling #
42 while(True):
43     value1 = int(input("Please input a first number to be divided: "))
44     value2 = int(input("Please input a second number to be divided: "))
45     try:
46         new_quotient = value1 / value2
47         with open(pickle_file, "wb") as combined_file:
48             new_values = pickle.dump(str(new_quotient), combined_file) # convert int to string
49             print("Data saved to file.")
50
51         with open(pickle_file, "rb") as load_quotient:
52             load_result = pickle.load(load_quotient)
53             print("This is the quotient from the unpickled file: ", load_result)
54         break
55     except ZeroDivisionError:
56         print("Numbers cannot be divided by 0. Please input a different number.")
57

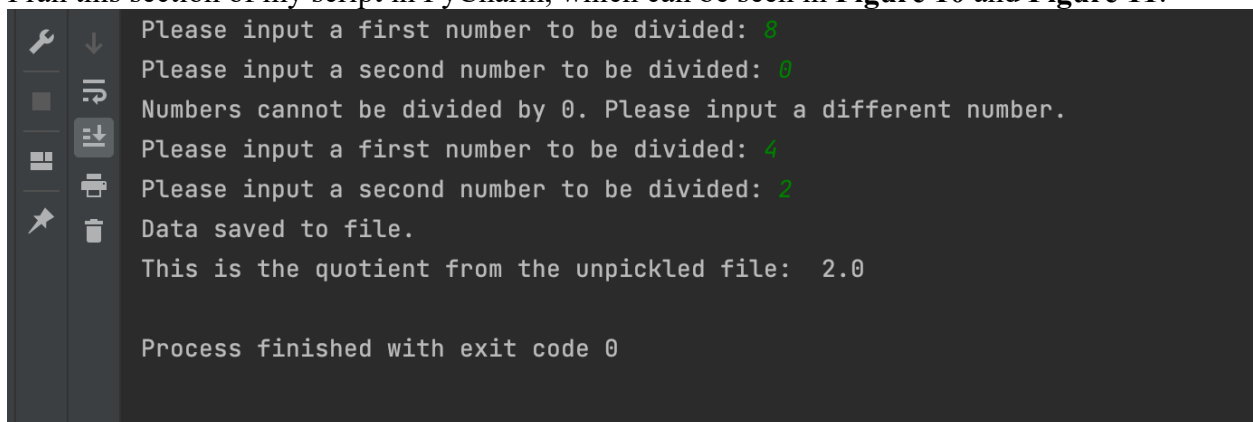
```

**Figure 9 — Combined try-except and pickling code**

For the second with-open statement, I had the binary file read through 'rb' and the data loaded back into the console. To ensure that the data had been unpickled (accessed through the load\_result) variable, I included a print statement that took in a string as well as the load\_result variable and printed it back to the user.

My except block used the Python exception ZeroDivisionError, which does not allow integers or floats to be divided by zero, to catch the errors. Since the except block is nested within the for-loop, once the print statement informing the user that numbers cannot be divided by zero is printed, the input prompts will run again.

I ran this section of my script in PyCharm, which can be seen in **Figure 10** and **Figure 11**:



```

Please input a first number to be divided: 8
Please input a second number to be divided: 0
Numbers cannot be divided by 0. Please input a different number.
Please input a first number to be divided: 4
Please input a second number to be divided: 2
Data saved to file.
This is the quotient from the unpickled file: 2.0

Process finished with exit code 0

```

**Figure 10 – Running the try-except and pickling section in PyCharm. The third line shows the print() statement that runs when a user tries to divide a number by 0. The seventh line shows the unpickled data printed out to the user.**

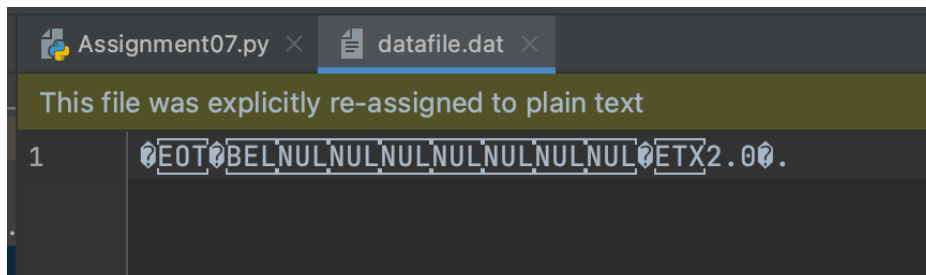


Figure 11 — The results of the try-except block: the saved quotient appears in the binary file, visible at the end of the line.

### 3.3 Running the Script in Command Shell:

I also ran my script in the command shell, which can be seen in **Figures 12** through **Figure 14**. **Figure 12** shows the full run of the script in the command shell, including where I added a new book title to the existing list, followed by testing the try-except and pickling sections.

```
Sebs-MacBook-Air:Assignment07 milaclendenning$ python3 Assignment07.py
Here is the pickled data from your file:
The Fifth Season
Tehanu
La Gesta dels Estels

Write the title of a book you'd like to read: Garcia Lorca: Obras Completas
Books To Read (New!):

The Fifth Season
Tehanu
La Gesta dels Estels
Garcia Lorca: Obras Completas
Please input a first number to be divided: 6
Please input a second number to be divided: 0
Numbers cannot be divided by 0. Please input a different number.
Please input a first number to be divided: 6
Please input a second number to be divided: 2
Data saved to file.
This is the quotient from the unpickled file: 3.0
```

Figure 12 — The script running in the Mac OS command shell

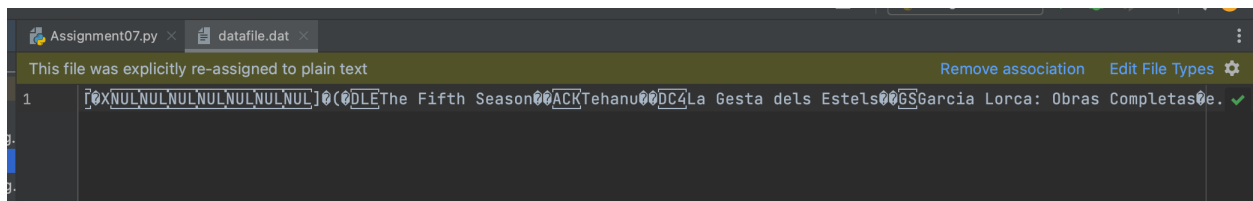
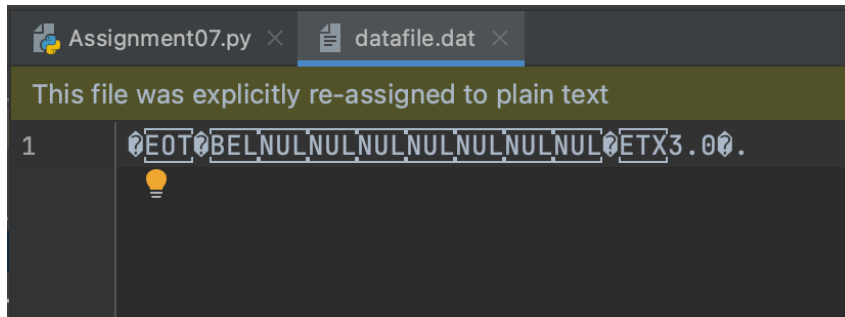


Figure 13 — The first result of the script having been run in the command shell, showing the book title 'Garcia Lorca: Obras Completas' having been saved to the binary file.



The screenshot shows a code editor with two tabs: 'Assignment07.py' and 'datafile.dat'. A green banner at the top states 'This file was explicitly re-assigned to plain text'. The editor displays a single line of text, which is a hexadecimal dump of a binary file. The text is: `1 \x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x00`. A lightbulb icon is visible below the text.

*Figure 14 — The last result of the script running in the command shell, showing that the quotient has been saved to the binary file since its result is not 0.*

#### 4. Summary:

This paper has described my pickling and error-handling script. It uses the pickle module from Python and the with-open() statements to dump and load data into a binary file and to present that unpickled, readable data back to the user. It also uses structured error handling to remind the user of errors and to make sure they've input the proper data type so that it can be properly saved to a file.