

Pickling and Error Handling in Python

1 Introduction

In this knowledge documentation, I cover a description of how I completed Assignment 7. Assignment 7 was a much more freeform and was centered around the topics of Structured Error Handling and Pickling.

Starter code was provided for this assignment which asked for the following to be done:

- For function *save_data_to_file* to be defined
- For function *read_data_from_file* to be defined
- Get an ID and name from the user, then store it in a list object
- Store a list object into a binary file
- Read the data saved in the binary file into a new list object and display the contents of the new list object.

2 Writing the Pseudocode

To create the code for Assignment 7 I first opened the starter code in PyCharm. Given that there was so little code in the starter code, it didn't take long to figure out what had already been written and how all the pieces interacted. Subsequently, I started drafting up pseudocode for these missing sections.

2.1 Processing

First, I began by drafting up the code for the Processing section, specifically the two functions included in the starter code: *save_data_to_file* and *read_data_from_file*.

```

# Processing ----- #
def save_data_to_file(file_name, list_of_data):
    """ Saves data from a List to a File

        :param file_name: (string) with name of file:
        :param list_of_data: (list) you want filled with file data:
        :return: nothing

    """
    pass # TODO: Add code here

# open File
# objFile = open("strFileName", "ab")

# dump
# pickle.dump(list_of_data, file_name)

# close file
# objFile.close()

```

Figure 1: Pseudocode for the first function in the script: `save_data_to_file`

Figure 1 shows us the first instance of pseudocode drafted up under the *Processing* section in the script. Here we see the first function `save_data_to_file` with pseudocode and comments drafted up underneath it. Given the name of the function from the starter code, while no other guidance was provided, the actions the function was to perform seemed clear. My drafted pseudocode included opening the file in append mode in binary format, although I could also see this function writing over all the data currently in the file. After opening the file, we use the `dump()` function imported from the `pickle` class to dump all the data from the list identified as an argument in the function. After dumping the data in binary format, we close the file.

I also drafted up comments for the function to summarize its purpose, its parameters, as well as what the function returns to make it clear for others reading the code to understand the function at a moment's glance.

```

def read_data_from_file(file_name):
    """ Reads data from File to a List

        :param file_name: (string) with name of file:
        :return: (list) of file data

    """
    pass # TODO: Add code here

# open file
#objFile = open("strFileName", "rb")
#objFileData = pickle.load(objFile) #load()
# only loads one row of data.<need to adjust so that it can load many rows?
#objFile.close()

```

Figure 2: Pseudocode added for the second function: `read_data_from_file`

Figure 2 shows us the pseudocode draft for the second function of the script in the *Processing* section of the code: `read_data_from_file`. Again, like with the function in Figure 1, there was not much guidance provided with the starter code. However, I felt like the purpose of the function was clear from its name. I drafted up some pseudocode to get started on tackling the task for this function. Given that we're reading data from a file, I determined that first I needed to open the file. After opening the file I simply defined a new list `objFileData` to load the data that was

pickled into. I did however note that this would simply load the first line that was pickled, not the entirety of data if there were multiple lines. I decided to tackle that later. Finally, I finished by closing the file and returning the list the previously pickled data was loaded onto.

Like with the first function, I wrote up a summary in the beginning of the function to summarize its purpose, the parameters it takes, as well as what it returns.

2.2 Presentation

The next section of the script is the Presentation. Here I was provided with three tasks by the starter code:

1. Get an ID and name from the user, then store it in a list object
2. Store a list object into a binary file
3. Read the data saved in the binary file into a new list object and display the contents of the new list object.

```
# Presentation ----- #  
  
# TODO: Get ID and NAME From user, then store it in a list object  
#get input ID  
#inputID = int(input("Enter an Id: "))  
  
#get input Name  
#inputName = str(input("Enter a Name: "))  
  
#save in list  
#lstCustomer= [inputID, inputName]
```

Figure 3: Pseudocode written for first TO DO of the Presentation Section

Figure 3 shows the pseudocode for the first task listed above. To get an ID and name from a user, I first defined two variables *inputID* and *inputName* and initialized them using the *input()* function. After assigning values to these two variables from the user's input, these variables are then saved into a list *lstCustomer*, declared under the Data section of the script.

```
# TODO: store the list object into a binary file  
#call save data to file  
# save_data_to_file(strFileName, lstCustomer)
```

Figure 4: Pseudocode for the second TO DO in the Presentation Section

The second task under the Presentation section of the script was to store the list, *lstCustomer*, into a binary file. To do so, I simply drafted pseudocode (along with actual Python) to call the previously defined function *save_list_to_file()*. The pseudocode drafted can be seen in Figure 4 where you can see the commented out Python code underneath the pseudocode.

```
# TODO: Read the data from the file into a new list object and display the contents
#call read_file_method
#allCustomers = read_data_from_file(strFileName)

#read the list
#print(allCustomers)
```

Figure 5: Pseudocode for final TO DO in the Presentation Section

The final drafted pseudocode was for the last task under the Presentation section of the script. This can be seen in Figure 5. The task here was to read the data from a file into a new list object and to then print out the contents of the list. To me it seemed clear that this was a place to call the *read_data_from_file* function previously. However, that function returns a list, so I needed to have a place to put the data from the function. This led me to initialize a new variable *allCustomers* with the data from calling the function *read_data_from_file*. After assigning data to that variable, I then needed to read the list, most likely with some sort of print function.

3 Writing the Code

After drafting up the pseudocode, as described in [2 Writing the Pseudocode](#) I started writing full statements out in Python.

3.1 Data

The first section of our code is *Data*. Here, variables and constants are initialized, if relevant to the section. Variables had already been declared in the data section and I did not see any need to add any more variables to the code than what had already been included.

3.2 Processing

Under the *Processing* section of the script is where the functions are defined for this script.

In our *Processing* section we have two functions:

- *Save_data_to_file(file_name, list_of_data)*
- *Read_data_from_file(file_name)*

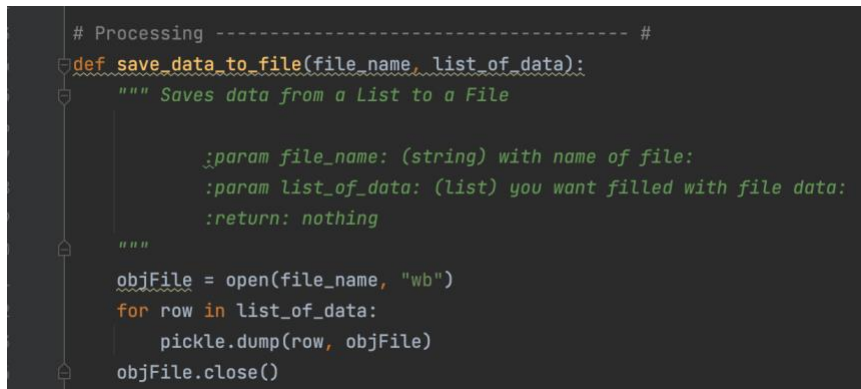
Both functions required editing for the assignment.

3.2.1 Saving Data to a File

The first function in the starter code was the function *save_data_to_file(file_name, list_of_data)*. This function takes two parameters: the file name we want to save data to and the list that holds the data we want to save.

The code for this function can be seen in Figure 6. Here, I opened the file in append mode in binary format, so that any data added would add the data to the end of the file rather than writing over the file. After opening the file, I made some additional edits to the pseudocode. Originally

there was simply a *dump()* function imported from the pickle class to dump all the data from the list identified as an argument in the function. However, I added a for loop to loop through the *list_of_data*. For each row, we perform the *dump()* function so that we are able to add multiple lines of data to the file of binary data if needed. I added this so that there is flexibility for the program to be able to store multiple lines of data in the initial list populated by the user. After dumping the data in binary format, we close the file.



```
# Processing ----- #
def save_data_to_file(file_name, list_of_data):
    """ Saves data from a List to a File

        :param file_name: (string) with name of file:
        :param list_of_data: (list) you want filled with file data:
        :return: nothing
    """
    objFile = open(file_name, "wb")
    for row in list_of_data:
        pickle.dump(row, objFile)
    objFile.close()
```

Figure 6: Code for function *save_data_to_file(file_name, list_of_data)*

Figure 6 shows the code for the function *save_data_to_file(file_name, list_of_data)*.

3.2.2 Reading Data from a File

The next function in the *Processing* section of the script is the function: *read_data_from_file(file_name)*. This function takes one parameter: the file name we want to read from.

Given that we're reading data from a file, we first open the file. After opening the file, a new list *objFileData* is defined to load the data that was pickled into. Like with the first function, I did some additional editing after drafting the pseudocode. I had noted that the pseudocode's structure of simply defining one list object with only one use of the *load()* function would only load the first line that was pickled, not the entirety of data if there were multiple lines.

I decided to address this with a while loop. Within this loop was a try-except structure for some structured error handling. The while loop loops through the file and loads line by line the data from the file to the list *objFileData*. At some point though the file will run out of data and Python will throw an EOFError, or End-of-File Error. Instead of allowing Python to stop my program with the error, I decided to use this as my trigger to break out of the while loop. So, under the *except* statement, I put the keyword *break* to break out of the while loop instead of halting the program when we run into the end of the file.

I finished by closing the file and returning the list the previously pickled data was loaded onto.

```

def read_data_from_file(file_name):
    """ Reads data from File to a List

        :param file_name: (string) with name of file:
        :return: (list) of file data
    """
    objFile = open("strFileName", "rb+")
    objFileData = []
    while True:
        try:
            objFileData.append(pickle.load(objFile))
        except EOFError:
            break
    objFile.close()
    return objFileData

```

Figure 7: Code for function `read_data_from_file(file_name)`

Figure 7 shows the code for the function `read_data_from_file(file_name)`.

To make it more user friendly, I then added some print lines to ensure that the user felt that they knew what was going on. These edits can be seen in Figure 8.

```

27 def read_data_from_file(file_name):
28     """ Reads data from File to a List
29
30         :param file_name: (string) with name of file:
31         :return: (list) of file data
32     """
33     print(f"Reading data from {file_name}...")
34     objFile = open(file_name, "rb+")
35     objFileData = []
36     while True:
37         try:
38             objFileData.append(pickle.load(objFile))
39         except EOFError:
40             print("No more data in file. Closing file...")
41             break
42     objFile.close()
43     return objFileData

```

Figure 8: Updated code for function `read_data_from_file(file_name)`

3.3 Presentation

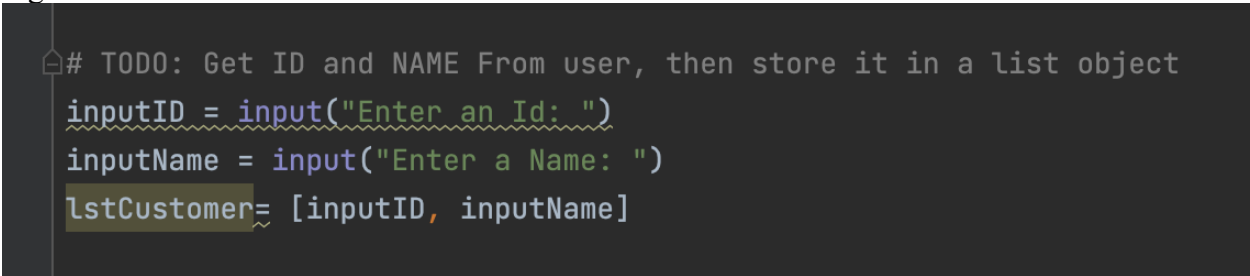
The next section of the script is the *Presentation* section. It is here where we see the code for the interaction between the user and the program. Pre-written in the starter script was three tasks:

- Get an ID and name from the user, then store it in a list object
- Store a list object into a binary file
- Read the data saved in the binary file into a new list object and display the contents of the new list object.

3.3.1 Getting input from the User

The first task under the presentation section was to get an ID and name from the user to store in a list object. First, I drafted up collecting the information from the user and then storing that information in a list.

Figure 9 shows the code for this draft.



```
# TODO: Get ID and NAME From user, then store it in a list object
inputID = input("Enter an Id: ")
inputName = input("Enter a Name: ")
lstCustomer = [inputID, inputName]
```

Figure 9: Code for the task to get an id and name from the user

However, I then decided to create another function to perform this task. At this point we are only performing this task once, but I could imagine wanting to have more flexibility in this code and wanting to get ID and name information from a user multiple times, in which having a function could make this easier.

The new function `get_id_and_name()` was placed under the *Processing* section of the code with the other functions. This function's purpose is to receive input from the user on the id and name they'd like to add. You can see the code for this function in Figure 10.

```
def get_id_and_name():
    try:
        inputID = int(input("Enter an Id: "))
    except ValueError: # except clause will only run if there is an error
        print("Please enter a number as an Id")
        inputID = int(input("Enter an Id: "))
    inputName = input("Enter a Name: ")
    return inputID, inputName
```

Figure 10: code for the function `get_id_and_name`

This function is very straightforward. First, we include a try and except statement. As we want the ID to be a number (or series of numbers), if the user enters any letters or other characters instead of halting the program, the try-except statement will prompt the user to enter the ID again, this time with numbers. A variable `inputID` is used to hold the user's input for an ID. Next the variable `inputName` is initialized with the user's input for a name. We then return both variables.

```
# TODO: Get ID and NAME From user, then store it in a list object
id, name = get_id_and_name()
lstCustomer = [id, name]
```

Figure 11: Code for task to get id and name from a user

To then tackle the task under the *Presentation* portion of the script, we then initialize variables `id` and `name` by calling the function `get_id_and_name()`. We then assign these values to the list `lstCustomer`. This process can be seen in Figure 11.

3.3.2 Storing the List to a Binary File

The next task in the *Presentation* portion of the script is to store the list to a binary file. Given that we've already set up a function to do that, `save_data_to_file(strFileName, lstCustomer)` we can use the function to accomplish this task.

```
# TODO: store the list object into a binary file
save_data_to_file(strFileName, lstCustomer)
```

Figure 12: Code for the task to store a list object into a binary file

Figure 12 shows the code for this task. Here we simply call the function and use the appropriate variables `strFileName` and `lstCustomer` to indicate which list we want saved into which file.

3.3.3 Reading and Displaying Contents of a Binary File

The final task of the TODOs given by the start script was to read the data from the file into a new list object and to display its contents. Figure 13 shows the initial draft of this code. Here we initialized the variable *allCustomers* by calling the function *read_data_from_file()*, passing the variable *strFileName* as an argument. This will populate the variable *allCustomers* with all of the IDs and names of the customers saved in the file. Next, in order to display the contents to the user, I printed the list. However simply printing the list does not make any adjustments to the output and will not provide the best output for the user.

```
# TODO: Read the data from the file into a new list object and display the contents
allCustomers = read_data_from_file(strFileName)
print(allCustomers) #customize how it's read
```

Figure 13: Initial draft of code for final TODO

In Figure 14 we see an updated piece of code for this task. Here I've added a for loop to loop through the list *allCustomers* and step by 2 so that we can print each ID and name together and not see duplicates of any items as we step through the list. This way each line will have the corresponding names and IDs of customers, resulting in an improved output for the user to view.

```
# TODO: Read the data from the file into a new list object and display the contents
allCustomers = read_data_from_file(strFileName)
print("The current data is: ")

for i in range(0, len(allCustomers), 2):
    print(allCustomers[i], allCustomers[i + 1])
```

Figure 14: Updated code for final TODO

4 Testing the Code

Now that my code was complete, I decided to test it out in two locations: via PyCharm and in my terminal.

4.1 PyCharm

To test the program on PyCharm, I simply clicked on Run in the upper command line to run the program I was editing.

```
Enter an Id: 1001
Enter a Name: Johanna
Reading data from AppData.dat...
No more data in file. Closing file...
The current data is:
1001 Johanna
```

Figure 15: Output from running the script

Figure 15 shows the program running as expected when the inputs are as expected (i.e. a number is inputted in for the ID).

```
Enter an Id: 1002
Enter a Name: Johanna2
Reading data from AppData.dat...
No more data in file. Closing file...
The current data is:
1001 Johanna
1002 Johanna2
```

Figure 16: Output when the script is re-run

We see in Figure 16 that the data is saved in the file and the new data is appended to the end of the file. When we then read the data from the file, all of the data is printed for the user.

```
Enter an Id: s
Please enter a number as an Id
Enter an Id1003
Enter a Name: Johanna3
Reading data from AppData.dat...
No more data in file. Closing file...
The current data is:
1001 Johanna
1002 Johanna2
1003 Johanna3
```

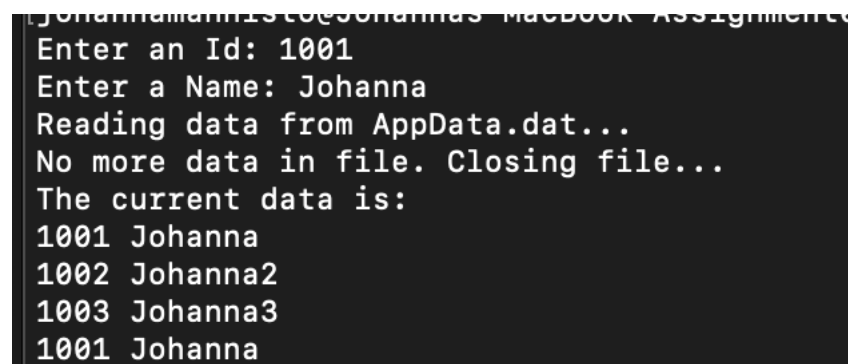
Figure 17: Output when error exception handling occurs in script

Figure 17 shows what the output is for the user when the error *ValueError* is thrown when a user inputs an ID. With the error handling set up in the script, as we see in Figure 17, it lets the user know that they should enter a number for the ID and prompts them to re-enter as expected.

Overall, the program ran as expected in PyCharm. We see the prompts we expect, the error exception handling is performing as desired, and the data is saved and read back for the user as intended.

4.2 Mac Terminal

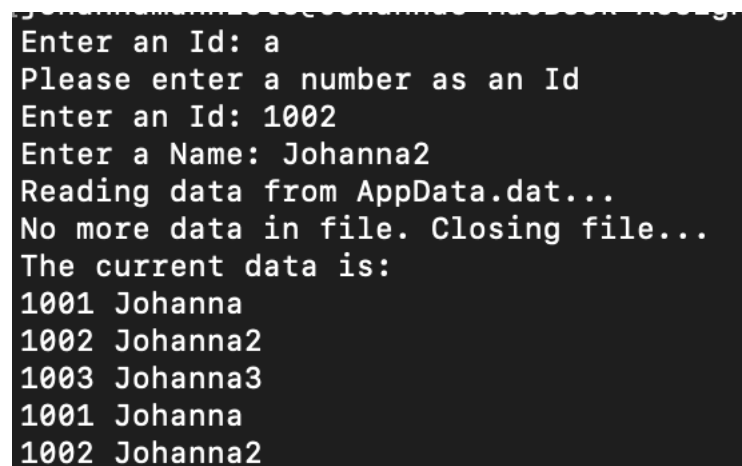
Everything also ran as expected in the Mac Terminal. The images below show the same outputs as seen in [4.1 PyCharm](#).

A screenshot of a Mac Terminal window with a black background and white text. The window title is 'JohannaMannisto@Johannus-MacBook-Air: Assignment 1'. The output shows the user entering '1001' for the ID and 'Johanna' for the name. The program then reads data from 'AppData.dat', reports 'No more data in file. Closing file...', and displays the current data: '1001 Johanna', '1002 Johanna2', '1003 Johanna3', and '1001 Johanna'.

```
JohannaMannisto@Johannus-MacBook-Air: Assignment 1
Enter an Id: 1001
Enter a Name: Johanna
Reading data from AppData.dat...
No more data in file. Closing file...
The current data is:
1001 Johanna
1002 Johanna2
1003 Johanna3
1001 Johanna
```

Figure 18: Output from running the script in Mac Terminal

Figure 18 demonstrates the same information as seen in Figure 15. Here, we see the program running as expected when the inputs are as expected (i.e., a number is inputted in for the ID).

A screenshot of a Mac Terminal window with a black background and white text. The window title is 'JohannaMannisto@Johannus-MacBook-Air: Assignment 1'. The output shows the user entering 'a' for the ID. The program prompts 'Please enter a number as an Id'. The user then enters '1002' for the ID and 'Johanna2' for the name. The program then reads data from 'AppData.dat', reports 'No more data in file. Closing file...', and displays the current data: '1001 Johanna', '1002 Johanna2', '1003 Johanna3', '1001 Johanna', and '1002 Johanna2'.

```
JohannaMannisto@Johannus-MacBook-Air: Assignment 1
Enter an Id: a
Please enter a number as an Id
Enter an Id: 1002
Enter a Name: Johanna2
Reading data from AppData.dat...
No more data in file. Closing file...
The current data is:
1001 Johanna
1002 Johanna2
1003 Johanna3
1001 Johanna
1002 Johanna2
```

Figure 19: Output when error exception handling occurs in script in Mac Terminal

Figure 19, like Figure 17, shows what the output is for the user when the error *ValueError* is thrown when a user inputs an ID. With the error handling set up in the script, as we see in Figure 19, it lets the user know that they should enter a number for the ID and prompts them to re-enter as expected.

5 Summary

In summary, Assignment 7 was a freeform assignment centered around the topics of Structured Error Handling and Pickling. Like with the previous assignments, I started off by writing pseudocode of what I thought I wanted the program to do, followed by writing out full code statements. Iterating through the assignment by reading through the script, writing out the pseudocode, and then going back to convert the pseudocode into actual Python allowed me time to process and think over ways to go through this assignment. I made the most changes to the code this time around after I wrote out the pseudocode.

Overall, the pickling process was very simple, as I feel comfortable calling functions and using them. The structured error handling was interesting to use, and I believe allows me to leverage certain errors to my advantage (for instance using the EOFError to break out of a while loop instead of needing to figure out how many lines are in the file in the first place to cycle through). There are many ways to use error handling to improve the experience for the user but to also use as elements in your script to move the program forward!