Phillip Thompson
5/30/23
IT FDN 110 A
Assignment 7
https://github.com/philthom10/IntroToProg-Python-Mod07
https://philthom10.github.io/IntroToProg-Python-Mod07/

# Assignment 7 – Travel Log – Files and Exceptions

## 1. Introduction

Module 07 this week focused on handling exceptions as well as more advanced use of files with Python. Specifically, the focus for this module and assignment was on the Pickling process for saving and loading objects to a binary file. I wrote a simple program that progresses through some steps capture user input, assigns the data to lists, pickles the lists to a binary file, then loads it. The program uses the example of logging accomplished and future travel locations for the user, but it is written in a way that it intended to just be used as an example for a user to show how pickling works. Ideally, I would have liked to have written a more sophisticated program that had its own useful function, but due to time constraints on this assignment, I chose to keep the program as just an example of how Pickling works. The program also uses handling on two custom exceptions in user input for the data.

## 2. Writing the Code

The code for this assignment was written in the PyCharm IDE, with a project structure similar to previous assignments, so I will not go into details in this report. The code for this program is in Assignment07.py.

This assignment was different than the previous in that the type of program was up to the student to choose, and there was no starter code or outline. I first started writing a much more complex program that added items as dictionaries, Pickled to binary files, which then would be called by key. After trying to tweak that code for some time, I decided that it had grown to a point that did not focus on the intent of this assignment, which is to demonstrate Pickles and handling exceptions in Python. I then decided to start again with a clearer picture of a simple program that demonstrated the basics of Pickles and exceptions.

The program has the user create three lists of travel locations as follows:

1) Locations the user has previously traveled to
2) Locations the user has near-term plans to travel to
3) “Bucket-list” travel locations that the user wishes to travel to in their lifetime

Listing 1, below, shows the framework, pseudo-code that I used to outline my program.

```
Import pickle

While loop to collect user input for three lists:
      Previously-visited travel locations
      Upcoming travel locations
      Bucket list travel locations
            *User exception to ensure input is not numeric
Use pickling to save lists to binary file
Load lists from binary file by unpickling
Display lists back to user
```

***Listing 1:*** *Framework Pseudo-Code for Program*

My outline did not initially contain the shelf method code, since that was a later addition upon reading more of the textbook.

The following sections of the knowledge document will describe certain features within the code. Due to its length, I will show figures or go into detail on every function, but will rather focus on those specific to the focus of this assignment – Pickling and exceptions.

## 2.1. Variable Declaration and Import Modules

The program starts by importing the pickle and shelve modules that I use later in the program. The pickle module allows me to "pickle a complex piece of data, like a list or dictionary, and save its entirety to a file". **(DAWSON, 200)**.

The global variables used in the program are then listed. While it is not necessary to declare all variables before using them in a line of code, I find it useful to do so in order to keep it clear which ones exist for reference later. See below (Figure 1) for the import module lines and variables. I ended up using two files with this program—one for the standard dump and load of lists to a binary file, and the other to demonstrate shelving lists.

```
8  import pickle
9  import shelve
10
11 # Data ------------------------------------------------------------------------ #
12 # defining initial variables
13 visited_list = []
14 planned_list = []
15 bucket_list = []
16 loaded_list = []
17 chosen_list = []
18 selected_key = None # string
19 type_choice = None  # string
20 str_file = "TravelList.dat"  # pickle file with travel list
21 shelf_file = "TravelListShelf.dat"  # separate file to use for shelf demo
22
```

**Figure 1:** *Module import and variable declaration*

## 2.2. Main Body

The main body of the code is shown below (Figure 2). The first While loop collects the user inputs to generate the three lists. It uses a simple method of user typing *"exit"* in order to break the loop. This is an area of program that I would make cleaner and more sophisticated with more time to work on it, but I chose to keep it simple in order to focus my time on the pickling and exception handling.

```
271 # Main Body ------------------------------------------------------------------------ #
272
273 IO.display_welcome_message()  # gives program intro message describing its purpose
274 while (True):
275     type_choice = IO.input_list_choice()  # prompts user for which list they want to update
276     if type_choice == "visited":
277         visited_list = IO.input_location(method=type_choice, location_list=visited_list)
278     elif type_choice == "planned":
279         planned_list = IO.input_location(method=type_choice, location_list=planned_list)
280     elif type_choice == "bucket list":
281         bucket_list = IO.input_location(method=type_choice, location_list=bucket_list)
282     elif type_choice == "exit":  # breaks from while loop if user types in exit
283         break
284     else:
285         print("Please type a valid option ('visited', 'planned', or 'bucket list'")
286 IO.display_created_lists(visited=visited_list, planned=planned_list, bucket=bucket_list)
287 IO.display_next_step()
288 Processor.dump_data_to_file(file_name=str_file, visited=visited_list, planned=planned_list, bucket=bucket_list)
289 IO.display_loading_steps()
290 loaded_visited, loaded_planned, loaded_bucket = Processor.load_data_from_file(file_name=str_file)
291 IO.display_created_lists(visited=loaded_visited, planned=loaded_planned, bucket=loaded_bucket)
292 IO.display_shelf_steps()
293 Processor.shelf_data_file(file=shelf_file, list_visit=visited_list, list_planned=planned_list, list_bucket=bucket_list)
294 selected_key = IO.input_shelf_key()  # uses function to receive selected key and assigns to variable
295 chosen_list = Processor.return_selected_shelf(file=shelf_file, key=selected_key)  # uses function to return list
296 IO.display_chosen_shelf(shelf=chosen_list, key=selected_key)  # formats chosen list for display back to user
297 IO.input_exit_prompt("You are now a pickle master! Use your power wisely. Press enter to exit the program.")
298
```

**Figure 2:** *Main body of code*

After the While loop, the code uses sequential function calls to display created lists, dump and load the lists to a binary files using pickling and shelving pickles, with some function calls describing the process added as necessary. I chose to just have one example of calling a shelf using a key input by the user, but that could easily be expanded into another loop that lets the user continue to call lists until the are done and break the loop. That would be more useful in a larger program with more data to call, so I just chose to show one example of it.

The details of the functions important to the use of the pickle function are highlighted later in this document.

## 2.3. Pickling via Dump and Load Functions – Processor Class

The processor class for this program has four functions, shown collapsed, below (Figure 3). These are the functions where the important pickle method steps take place, so I will detail them further.


**Figure 3:** *Functions within Processor class*

### 2.3.1 Pickling Lists to a Binary File: *dump_data_to_file()* Function

The first Processor class function called in the main body is *dump_data_to_file()*. The code for this function is shown below (Figure 4).

The function receives the file name to be written to as a parameter, as well as the three lists that the user has previously created as additional parameters. The *file_name* parameter had the argument of the binary file name "TravelList.dat" passed to it. This is different than the .txt extension files we have used in this course previously. This is because "Pickled objects must be stored in a binary file – they can't be stored in a text file". **(DAWSON, 201).** This functions opens the file using the access mode *wb*, which writes to the binary file, overwriting its contents. The *wb* mode also creates the binary file if it doesn't exist. This is as described in Table 7.3 of **(DAWSON, 201).**

Because this is a simple program to demonstrate pickling and error handling, each time it is run by a user, the file will be overwritten with new data. This is as-designed, so that it is not confusing to the user to have existing data strings already in the file. A more advanced and useful program would allow the user to load existing strings and add to them.

```
51      @staticmethod
52      def dump_data_to_file(file_name, visited, planned, bucket):
53          """ Writes data from a list of dictionary rows to binary file
54
55          :param file_name: (string) with name of file:
56          :param visited: (list) of locations visited:
57          :param planned: (list) of locations planned to visit soon:
58          :param bucket: (list) of bucket-list locations to visit:
59          :return: (none)
60          """
61          file = open(file_name, "wb")  # opens file with passed in file_name parameter
62          #dumps the three lists to the file one-by-one
63          pickle.dump(visited, file)  # dump saves one list at a time
64          pickle.dump(planned, file)  # dump saves one list at a time
65          pickle.dump(bucket, file)  # dump saves one list at a time
66          file.close()
```

**Figure 4:** *Code for dump_data_to_file() function*

The *pickle.dump()* function is what stores the lists in the binary file. It only stores one data item at a time, so it is called three times in the function. Each call of the function requires the two arguments of the item to be pickled (list) as well as the file name, so the higher-level function parameters are used as applicable. When this *dump_data_to_file()* function is done running, the three lists are successfully stored on the binary file.

### 2.3.2 Loading Lists from Binary File: *load_data_from_file()* Function

The next step of the program is to load the lists that were just stored by the user from the binary file. This is accomplished using the *load_data_from_file()* Function, with the code shown below (Figure 5).

```
68      @staticmethod
69      def load_data_from_file(file_name):
70          """" Reads data from a pickle file
71          :param file_name: (string) with name of file:
72          :return: (list, list, list) of locations
73          """
74          file = open(file_name, "rb")
75          # loads the lists back from binary file, one row of data at a time
76          visited_locs = pickle.load(file)
77          planned_locs = pickle.load(file)
78          bucket_locs = pickle.load(file)
79          file.close()
80          return visited_locs, planned_locs, bucket_locs
```

**Figure 5:** *Code for load_data_from_file() function*

This time, the file is opened with the *rb* mode, which reads from a binary file. Similar to the *pickle.dump()* function, the *pickle.load()* function only loads one piece of data item at a time, so I call it three times and assign those loaded lists to local variables that are returned by the *load_data_from_file()* function. Note that the *pickle.load()* "function only takes one argument: the file from which to load the next pickled object". **(Dawson, 202).**

## 2.4. Using a Shelf to Store Pickled Data – Processor Class

The other two functions within this program's Processor class are the two for storing and loading data from a binary file using the shelf method. The benefit of using the shelf method is that the lists are paired with a key that can be used to load them, rather than just the sequential loading that is done using the *pickle.load()* function.

### 2.4.1 Storing Pickled Lists on Shelf in Binary File: *shelf_data_file()* Function

First, I used the *shelf_data_file()* function as shown below (Figure 6).

```
35      @staticmethod
36      def shelf_data_file(file, list_visit, list_planned, list_bucket):
37          """ Shelves data to file
38          :param: file (string) name of file
39          :param: list_visit (list) of visited locations
40          :param: list_planned (list) of planned locations
41          :param: list_bucket (list) of bucket list locations
42
43          :return: (none)
44          """
45          s = shelve.open(file)
46          s["visited"] = list_visit  # first list stored on shelf with key  "visited"
47          s["planned"] = list_planned  # second list stored on shelf with key  "planned"
48          s["bucket"] = list_bucket   # third list stored on shelf with key  "bucket"
49          s.sync()  # make sure data is synced
50
```

**Figure 6:** *Code for shelf_data_file() function*

This function receives the other file name "TravelListShelf.dat" as an argument, as well as the three lists that the user has previously created. Then the function creates a shelf assigned to local variable *s* using the *shelve.open()* function on the file. Each list is then are added to the shelf with a corresponding key. This follows the method shown in the example in the class textbook **(Dawson, 203).**

### 2.4.2 Loading Pickled Lists Using a Shelf: *shelf_data_file()* Function

Finally, the *shelf_data_file()* function is used to retrieve the lists from the binary file. See below (Figure 7) for the code of this function. The file name and the key selected by the user are passed as parameters into this function, then if / elif statements are used to compare the user-selected key with the keys in shelf of

the pickled lists. The matching list is then returned and displayed to the user. I did not specify an access mode argument in the *shelve.open()* function, as the default *c* mode to read or write, as described in table 7.5 of **(DAWSON, 203)** was sufficient for the needs of my program.

```
82      @staticmethod
83      def return_selected_shelf(file, key):
84          """ Calls list shelf from file using user-chosen key
85          :param file: (string) with name of file
86          :param key: (string) with chosen shelf key
87          :return: (list) of locations
88          """
89          s = shelve.open(file)  # opens the file with shelf method, assigns shelf to local variable
90          if key == "visited":
91              shelf_list = s["visited"]  # assigns the list from called key to local variable to return
92          elif key == "planned":
93              shelf_list = s["planned"]
94          elif key == "bucket":
95              shelf_list = s["bucket"]
96          s.close()
97          return shelf_list
```

**Figure 7:** *Code for return_selected_shelf() function*

## 2.5. Exception Handling Within IO Class

The IO class of this program contains ten total custom function, as shown with details collapsed below (Figure 8). Several of these functions simply print instructions or descriptions of the process as a training aid, so I will not detail those functions in this document. The two IO functions that I will detail demonstrate error handling / exceptions.

(space intentionally left blank)

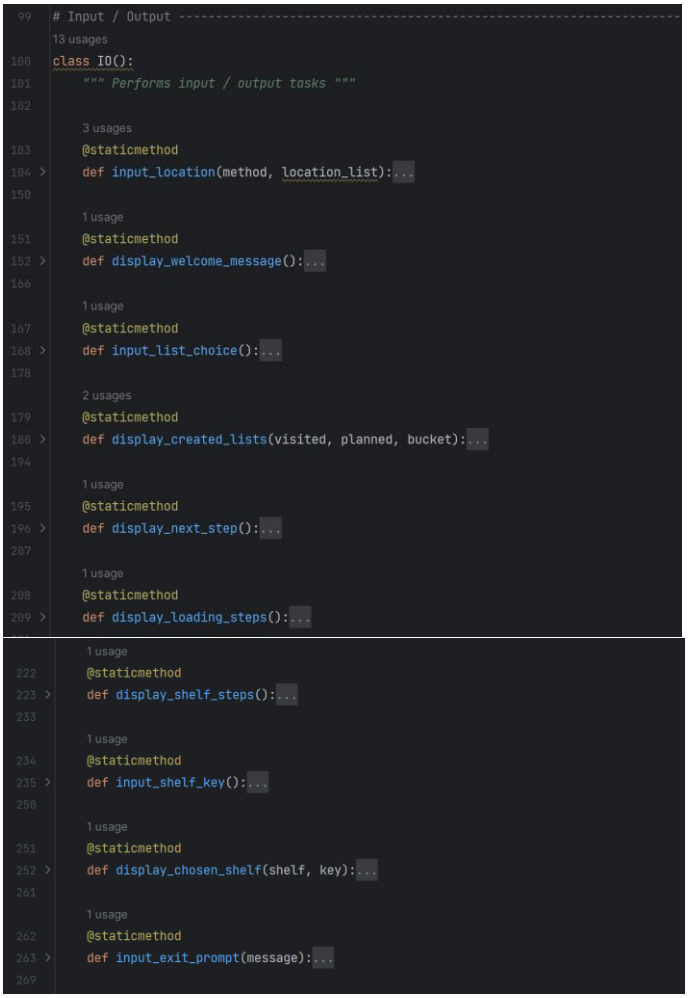


**Figure 8:** *IO() class functions*

### 2.5.1 Exception in *input_location()* Function

The purpose of the *input_location()* function is to take the selection of the list that the user wishes to update, passed as a parameter, and prompt for / capture continues input of vacation locations until the user types “done” to stop the entry. The function then returns a list created from the user input for that specific vacation category. The code for this function is show below (Figure 9).

```python
    @staticmethod
    def input_location(method, location_list):
        """ Captures user input of types of locations
        :param method: (string) user chosen method for which list to edit:
        :return: (list) of locations
        """
        location = None
        # based on previous user input choice of which list to update, has different wording prompts for data entry
        if method == "visited":
            print("* Note: Enter 'done' as the location to finish your list *")  # option for user to stop entry for that list
            print()  # new line for looks
            while (location != "Done"):
                try:
                    location = input("Enter a location that you have previously traveled to: ").title().strip()
                    if location.isnumeric() == True:
                        raise NumericLocationException
                    else:
                        location_list.append(location)
                except NumericLocationException:
                    print("  ** You entered a numeric value. Please enter a valid location.")
        elif method == "planned":
            print("* Note: Enter 'done' as the location to finish your list *")  # option for user to stop entry for that list
            print()  # new line for looks
            while (location != "Done"):
                try:
                    location = input("Enter a location that you plan to travel to in the next two years: ").title().strip()
                    if location.isnumeric() == True:
                        raise NumericLocationException
                    else:
                        location_list.append(location)
                except NumericLocationException:
                    print("  ** You entered a numeric value. Please enter a valid location.")
        elif method == "bucket list":
            print("* Note: Enter 'done' as the location to finish your list *")  # option for user to stop entry for that list
            print()  # new line for looks
            while (location != "Done"):
                try:
                    location = input("Enter a location that is on your bucket-list to travel to in your lifetime: ").title().strip()
                    if location.isnumeric() == True:
                        raise NumericLocationException
                    else:
                        location_list.append(location)
                except NumericLocationException:
                    print("  ** You entered a numeric value. Please enter a valid location.")
        location_list = location_list[:-1]  # strips last item from list to remove "done" from list before returning it
        return location_list
```

**Figure 9:** *input_location() function code*

I used this function as a chance to demonstrate exception handling by raising a custom exception if the user inputs a numeric value. I defined the custom exception *NumericLocationException* at the beginning of my program, see below (Figure 10). Typically these custom exceptions would be in a separate file, but for the purposes of this assignment, I included them in the Assignment07.py file.

```
# Exceptions / Error Handling------------------------------------------------------------ #
6 usages
class NumericLocationException(Exception):  # custom exception to be used later
    """Raised when input is only numeric."""
    pass
```

**Figure 10:** Custom *NumericLocationException* class

The exception is handled using the *try* statement with the *except* clause **(Dawson, 206).** Under the try statement, the input of vacation location by the user is captured, and the *isnumeric()* function is used to evaluate if the user entry is a numeric value. If the input is indeed numeric, the custom exception *NumericLocationException* is raised. If the exception is raised, the *except* clause then prints feedback to the user letting them know it was an invalid input. Since this try-except section is nested under the *while* loop, it provides the *except* feedback and prompts for user input until a valid input is received.

### 2.5.2 Exception in *input_shelf_key()* Function

The second function with exception handling is the *input_shelf_key()* function. This function prompts the user to input a key word that is used to retrieve a pickled list, as detailed in section 2.3.2 of this paper. The code for this function is shown below (Figure 11).

```
    @staticmethod
    def input_shelf_key():
        """ Collects input from user on key to call shelf list from

        :return: key (string) with selected key from user
        """
        key = None  # defining key with no value prior to while loop
        while key != "visited" and key != "planned" and key != "bucket":  # loops as long as key is invalid
            try:
                key = input("Which list would you like to see ('visited', 'planned', 'bucket')? ").lower().strip()
                if key != "visited" and key != "planned" and key != "bucket":  # compares against valid choices
                    raise InvalidChoiceException   # raises exception class if invalid input is received
            except InvalidChoiceException:
                print("Invalid key selection. Must be one of listed options.")
        print()  # extra line for looks
        return key

```

**Figure 11:** *input_shelf_key()* function code

While the exception handling detailed in the previous section for *input_location()* was a good demonstration, the exception handling in this *input_shelf_key()* provides a practical use. It is important for the user to input the key exactly as desired, else it will not match the key of the shelf. I used another custom exception class *InvalidChoiceException* for this. See below (Figure 12).

```
27  class InvalidChoiceException(Exception):   # custom exception to be used later
28      """Raised when input is invalid."""
29      pass
```

**Figure 12:** Custom *InvalidChoiceException* class

Comparison operators are used in the try block to compare the user input against the three acceptable values. If the input does not match any of those, *InvalidChoiceException* is raised, and the user is printed feedback that the input is not acceptable. The first several times of me running the code with this try-except block of code caused errors that were not easily deciphered using the error description. I used PyCharm's debug function to determine that while the exception was being raised, the function was still trying to return a key value, causing the program to crash. I realized that I needed a While loop to prompt the user for input again. The resulting While loop code can be seen above (Figure 11). Once that was added, the program ran successfully, and my program was complete.

## 3. Running the Program

The program eventually ran successfully both in PyCharm (Figure 13) as well as the Windows command window (Figure 14). The display of instructions and descriptions back to the user is not perfect, and with more time I would improve the overall look and feel of the program. However, it sufficiently shows pickling and exception handling.

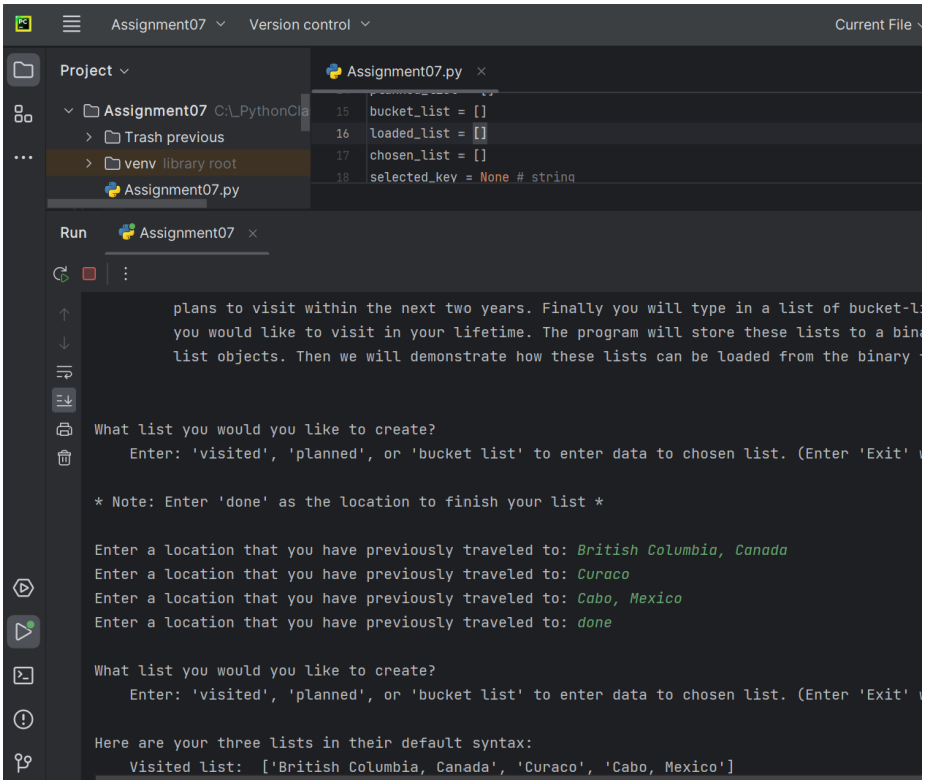


***Figure 13: Running Program in PyCharm***

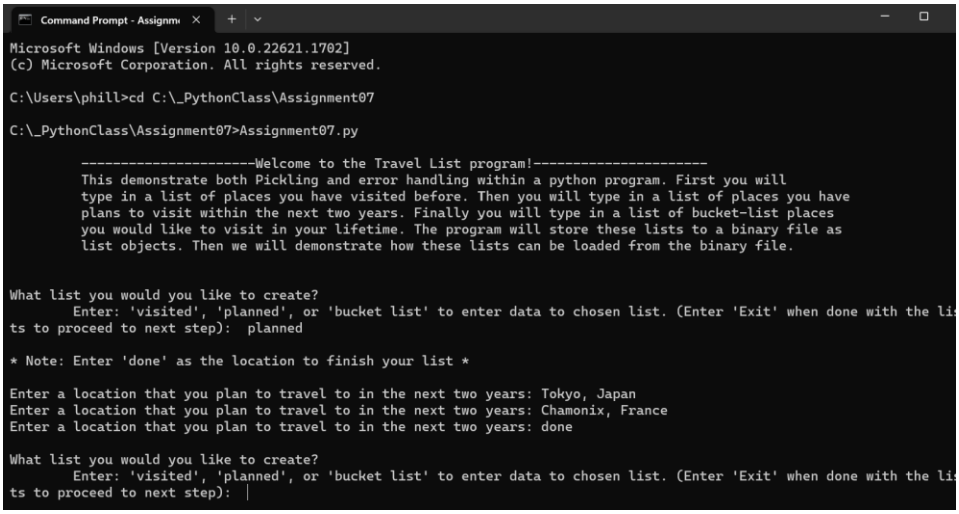


***Figure 14: Running Program in Windows command window***

I verified that the data was saved in the *.dat* files for both methods, in the correct relative directory. See below (Figure 15) for an image of the *TravelList.dat* file open in Notepad.

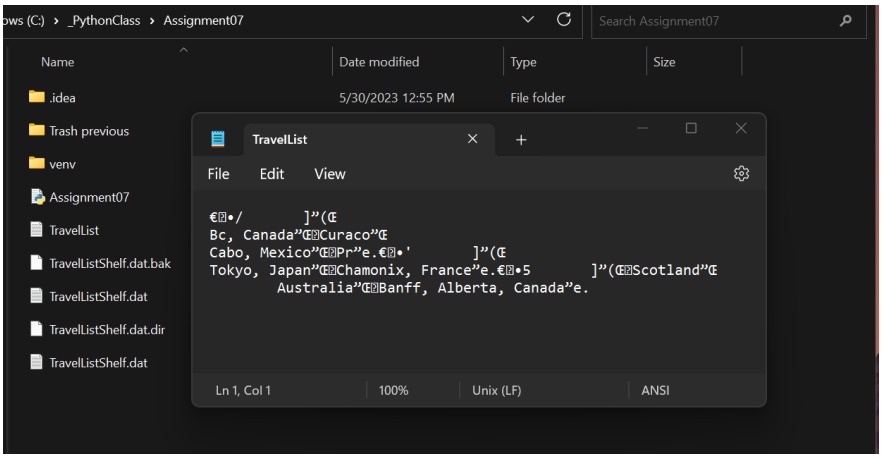


***Figure 15:** .dat files created in folder and opened*

## 4. Summary

This assignment demonstrated the Pickle method to store and load complex data to a binary file. It also was my first time using exception handling. The usefulness of creating custom exceptions should lead to a better experience for controlling user input and other things without more lines of code to handle it. The troubleshooting of the exception handling in the *input_shelf_key()* provided another chance to use debugging in PyCharm.

Having more freedom to decide on the type of program was interesting, but it led me to some wasted time in writing an initial program that did not make sense for demonstrating pickling. This was a good

learning experience to sort out the idea further in the project-planning phase, not just for how it is structured, but if it makes sense for what I am trying to accomplish.

## References

Dawson, Michael. *Python Programming for the Absolute Beginner*, Third Edition, Cengage Learning, 2010

"Python Pickle - Working with Pickle in Python." educba.com, www.educba.com/python-pickle/. Accessed 29 May 2023.

"Try and Except in Python - Python Basics." pythonbasics.org, pythonbasics.org/try-except/. Accessed 30 May 2023