Linda Shmait
8/24/22
Foundations of Programming: Python
Assignment 07
https://github.com/lzs425/IntroToProg-Python-Mod07

History Script: Using Pickle and Error Handling

Introduction

This article will review a script created to provide a user a historical fact or fun piece of information based on a month and day input. It is built on a very similar structure to Assignments 5 and 6, with a menu of options the user can navigate through based on different input values.

Pickling was used to store the data externally. The script requires the corresponding .dat file, "EventsbyDate.dat" for unpickling and pickling the date and event data. The script requires several inputs from the user, and I used error handling to structure the responses in a way that the user would be required to use inputs that the script could handle.

The result is a script in which you can view information by month and day, remove and replace data, save it externally in a binary file, and call that data back in upon starting the program next time. The results can be seen in Figure 1 below, with actual output depending on user input.

```
        This program shares a historical tidbit by day, sourced primarily from "A Leap Year of Great Stories,
        366 from History for Every Day of the Year" by WB Marsh & Bruce Carrick, as well as some general classics!

        Menu of Options

        1) View all the Data
        2) Input your month and day to get a fact!
        3) Input and save your own fact of the day!
        4) Save your data
        5) Exit Program


Which option would you like to perform? [1 to 5] - 2
Please input the numeric month number of the date: 1
Got it! Great month.
Please input the numeric day of the month: 2
Got it! Let's see if that date exists...You will be routed to the menu if you're close but no cigar
End of the Reconquista after Ferdinand and Isabella take Granada in 1492

        This program shares a historical tidbit by day, sourced primarily from "A Leap Year of Great Stories,
        366 from History for Every Day of the Year" by WB Marsh & Bruce Carrick, as well as some general classics!

        Menu of Options

        1) View all the Data
        2) Input your month and day to get a fact!
        3) Input and save your own fact of the day!
        4) Save your data
        5) Exit Program


Which option would you like to perform? [1 to 5] - 3
Please input the numeric month number of the date: 1
Got it! Great month.
Please input the numeric day of the month: 2
Got it! Let's see if that date exists...You will be routed to the menu if you're close but no cigar
Current entry:
End of the Reconquista after Ferdinand and Isabella take Granada in 1492
```

Figure 1. Completed Script Scenario

Script

Before building the script, I did have to write out my pseudocode to see what functions I needed to accomplish the tasks I wanted for the script. I used similar organization to previous assignments, so the script consisted of Processing, Input/Output section, and Main sections. The Processing section contains actions that do not require user input, though unlike previously there are some output messages associated with error handling to notify the user if there is a back-end error around the files or date matching. The Input/Output section has the menu to display, the menu choice, month and day inputs, table printing, a method to ask the user if they want to add/replace an event, and a method to intake that message. All of these I/O functions have some form of error handling.

Processing Section

The processing section consists of the class Processor, which contained data to save a table to a file in binary (pickling), read binary data from a file and store it in a table (unpickling), identifying the event associated with a specific month and day from that table, and adding an event (replacing the old one).

The two methods in Processing that did not use pickle were the event_by_date(month, day, list_of_rows) and add_event_to_date(month, day, new_event, list_of_rows). The only new code in them was related to error handling and will be discussed later in this essay.

Pickle Module

Two methods in the Processing section utilize the Pickle module. While the data I am storing is not complex, that is the main purpose of pickling: to save space by preserving your data in binary form. Looking at the Python docs, it is described as a "module [that] implements binary protocols for serializing and de-serializing a Python object structure" (pickle — Python object serialization, pickle — Python object serialization — Python 3.10.6 documentation, 8/20/22)(External site) . The article essentially describes how different objects, like lists and dictionaries, can be pickled and "unpickled". There are different modes for unpickling, or essentially translating the binary back into the original form. Saving pickle data is done through the dump() function and unpickling through the load() function.

Like the non-binary .txt files we've worked with previously, the access modes allow for reading, writing, appending, reading/writing, and appending/reading combinations. The mode is determined when you open the file. Figure 2 on the next page shows how I "pickle" and "unpickle" my data.

Pickled data also allows you to use shelving. It will open a pickled file in different modes ('c' for read/write, 'n' for new file to read/write, 'r' to read', and 'w' to write. Each of these modes has create/edit attributes that can be reviewed on page 204 of our textbook (Dawson, Python Programming for Absolute Beginners, Cengage Learning, 2010). Shelving and allow you to convert the data into lists and easily assigns names to columns. However, as I was recycling code, I did not utilize shelving, only pickling.

```python
    @staticmethod
    def save_data_to_file(file_name, list_of_rows):
        """Writes the data from list_of_rows into a binary file
        :param file_name- the file being written to
        :param list_of_rows- the data table being written to the file
        """

        file_obj_pickle = open(file_name, "wb")
        pickle.dump(list_of_rows, file_obj_pickle)
        file_obj_pickle.close()

    @staticmethod
    def read_data_from_binary_file(file_name):
        """Reads data from file_name - a binary file- and writes it to a table
        :param file_name- the file being read
        :return table of data unpickled"""
        try:
            file = open(file_name, "rb")
            table = pickle.load(file)
            file.close()
        except (FileNotFoundError):
            print("File issue, did you save your pickle file in the same location as the code?")
        return table
```

Figure 2. Pickling and Unpickling Methods

The first method save_data_to_file requires the file name and the table to which the file info will be "unpickled" and stored. I call read_data_from_binary_file(file_name) first in order to load the existing binary event data and "unpickle" it using the load() function into a local table. Only upon user request (Option 4 in the menu) do I write the data into my binary external file, in this case "EventsbyDate.dat" using the event_by_date(month, day, list_of_rows) function.

Input/Output

The Input/Output section is where I collect information from the user and use data validation to ensure the inputs will allow the program to continue. There are methods to call up the menu (essentially borrowed from the previous assignment), capture the menu option choice, get the user's desired month and day, print the entire data table, ask the user if they want to input a new event, and capture that new event.

Error Handling

All but the first method to print the menu use some form of data validation. During my reading, I saw try/except error handling, raising an exception, using try/except/else, and logic error handling with if/try/else combinations. I found the article "Errors and Exceptions" helpful in running through the difference between different types of errors and exceptions ("Errors and Exceptions", https://docs.python.org/3/tutorial/errors.html, 8/22/22). There were also some good examples of error handling in stackoverflow in Figure 3. I emulated that code when building my function to unpickle the data.

```python
while True:
    prompt = input("\n Hello to Sudoku valitator,"
    "\n \n Please type in the path to your file and press 'Enter': ")
    try:
        sudoku = open(prompt, 'r').readlines()
    except FileNotFoundError:
        print("Wrong file or file path")
    else:
        break
```

Figure 3. stackoverflow Example of Error Handling with Try/Except

I also took a look at the article “Python- Error Types” that reviewed the common Python error types with descriptions a bit more extensively than our Python Programming for the Absolute Beginner. While the list was extensive, I primarily used KeyError and ValueError. For example, I used KeyError in case there is no match between months/days entered and the data table (Python_Error Types, https://www.tutorialsteacher.com/python/error-types-in-python, 8/24/22).

Examples of Error Handling in Script

Once I learned a bit more about error handling, I tried refining my script to put more boundaries on entries and provide the user more information regarding faulty or undesirable inputs. I used try/except with ValueError and upper and lower bounds on that input value to ensure that the menu responses were limited to numbers 1-5. This can be seen in Figure 4 below.

```
129         choice_int = 0
130         while (choice_int<1) or (choice_int>5):
131             try:
132                 choice = str(input("Which option would you like to perform? [1 to 5] - ")).strip()
133                 choice_int = int(choice)
134             except ValueError:
135                 print("Please only enter 1-5")
136         return choice
```

Figure 4. Try/Except in Script

In other functions, I tried using logic to address potential errors. When getting month and day inputs for example, the table had them stored as strings, but to limit the range and eliminate the potential for a user to input leading zeros, I used casting as well as try/except. I also learned not to have my while statements within my try/except functions if I wanted them to run properly… the hard way. Figure 5 shows the end result of the day input code.

```
156     def input_day():
157         """function to capture day number from user and ensure it is in the right format
158         :return:day_str"""
159         day_int = 0
160         while (day_int < 1) or (day_int > 31):
161             try:
162                 day_input_str = input("Please input the numeric day of the month: ")
163                 day_int = int(day_input_str)
164                 day_str = str(day_int) #ensure no leading 0s
165                 if(1<=day_int<=31):
166                     print("Got it! Let's see if that date exists..."
167                           "You will be routed to the menu if you're close but no cigar")
168                 else:
169                     print("Is this a real day? Please only enter a valid day date,"
170                           "and remember the rhyme:"
171                           "'Thirty days has September, April, June, and November. "
172                           "All the rest have 31'...except Feb. You're on your own.)")
173             except ValueError:
174                 print("Is this a real day? Please only enter a valid day date for your selected month.")
175         return day_str
```

Figure 5. Day Input Data Validation and Error Handling

There were many options I could have taken, including raising custom error messages, but could not figure out how to call on it without the result exiting the program and ended up leaning on other error handling processes. This will be an area of additional learning for me.I used similar methodologies to if/else/except across the different I/O methods.

Main

After completing the different classes and methods needed, I called them within the appropriate sections of the main code. I first called in my function to unpickle data and stored it in table_list. I then entered the while(True) structure with if statements around the menu selection based on the user's input choice.

An example of one of the menu selections, the user requesting to input a new event from the main menu, is below in Figure 6. By combining the different methods, I was able to run through the menu options in the main section.

```
elif choice_str == '3':  # Input new event to table
    month_str = IO.input_month()
    day_str = IO.input_day()
    print("Current entry:")
    hist_event_str = Processor.event_by_date(month=month_str, day=day_str, list_of_rows=table_lst)
    print(hist_event_str) #show current event for that date
    response_event_req = IO.new_event_request() #make sure they want to proceed
    if(response_event_req == 'y'):
        event = IO.input_new_event(month=month_str, day=day_str, list_of_rows=table_lst,new_event_response=response_event_req)
        Processor.add_event_to_date(month=month_str, day=day_str, new_event=event, list_of_rows=table_lst)
        print("Ok, this event has been added!")
    else:
        print("Back to main menu!")
    continue  # to show the menu
```

Figure 6. Main Code Sub-Section Example

Testing

This script required extensive testing. At some point, I was unable to print my table because I had allowed entries other than a string, and used the program feedback to address the issue by forcing each element of the dictionary to print as string.

```
12/30 - Grigori Rasputin assassinated in St Petersburg in 1916
12/31 -
Traceback (most recent call last):
  File "C:\_PythonClass\Assignment07\Assignment07.py", line 207, in <module>
    IO.print_the_year(table_lst)
  File "C:\_PythonClass\Assignment07\Assignment07.py", line 166, in print_the_year
    print(row["Month"] + "/" + row["Day"] + " - " + row["Event"])
TypeError: can only concatenate str (not "list") to str

Process finished with exit code 1
```

Figure 7. Troubleshooting During Development

There were many other issues along the way, from indent to missing colons, to closing gaps in my logic. My last step was to test the code in the Command Prompt. An example can be seen below in Figure 8.

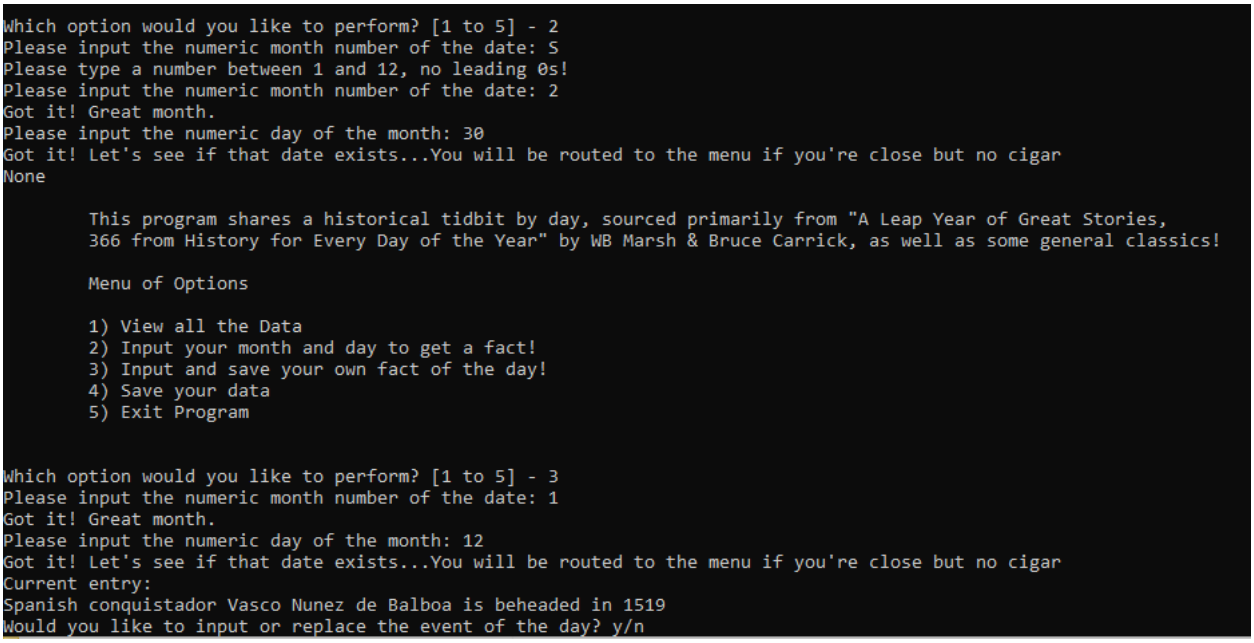

Figure 8. Command Prompt Testing

Summary

This script built on previous assignments while incorporating pickling and error handling to more efficiently store data and ensure the program could run smoothly. It successfully reads the binary data file to pull in dates and events, then allows the user to navigate through the menu to view and input the information they would like to see- or simply view a bit of history.