Edwin Kintu-Lubowa

November 9, 2024

Foundations of Programming: Python

Assignment05

https://github.com/EdwinLubowa/Python110-Fall2024

Advanced Collections and Error Handling

Introduction

This week, I learned about lists and dictionaries and their inherent, fundamental differences; reading from and writing data to a dictionary; how Python's JSON module could simplify data handling; structured error handling and the use of the try-except construct. This week's assignment was a confluence of all these concepts: it demonstrated the use of dictionaries, files and structured error handling using the try-except construct. In this knowledge document, I will chronicle the concepts I learned, and how I used them to complete the assignment.

File handling and the JSON module

Previous to this week, I learned how to read from and write to files in Python. The first collection of data I wrote to the file was a formatted list that was saved to a CVS file as a comma separated list of values. In the Python, the procedure for writing to the file was structured and intuitive:

- I used the open() function to open the file in the desired mode; "w", "a", — for "write", "append" respectively
- I used the write() function to write data to the file
- Then finally closed the file using the close() function to close the file

Alongside writing data to a file, I also learned, in previous weeks how to read data from a file, which process was, also, as structured and intuitive as reading data from the file:

- I used the open() function to open the file with "r" read permission
- I used the readlines() function to read data from the file
- Then finally closed the file using the close() function.

In this week's assignment, I used the same format to read data from the file. However, in the assignment, I was working primarily with "dictionaries" as referenced by the key-value pairing in the "student_data" dictionary below. The code got a bit more complex and longer to

accomplish the task of iterating each row of data from file and appending it to “students” list using the append() function. (Figure 1)

```
36
37  file_obj = open(FILE_NAME, "r")
38  for row in file_obj.readlines():
39      # Transform the data from the file
40      student_data = row.split(",")
41      student_data = {"FirstName": student_data[0],
42                      "LastName": student_data[1],
43                      "GPA": float(student_data[2].strip())}
44      # Load it into the collection
45      students.append(student_data)
46  file_obj.close()
47
```

Figure 1: reading data from file using readline() function

I also saved data to a file, this week, using the write() function, this, too, was complicated by the need to iterate through the “students” table and write each row into the file; then close the file afterwards. In the code below, each row of data is a dictionary as referenced by the key-value pairing. (Figure 2)

```
92
93          file_obj = open(FILE_NAME, "w")
94          for student in students:
95              file_obj.write(f"{student["FirstName"]},{student["LastName"]},{student["GPA"]}\n")
96          file_obj.close()
97          print("Data Saved!")
98          continue
99
```

Figure 2: saving a dictionary row to a file

The process of reading data from the file got a lot easier with the importation of the JSON module into the Python script. What took nine lines of code to process using the previous method was, now, achieved in three lines of code. In this code, I opened the file, in read mode, then used the “json.load()” function to read the file data into a list of dictionary rows “students”; and closed the file using the close() function. (Figure 3)

```
31
32  # When the program starts, read the file data into a list of dictionary rows (table)
33  file_obj = open(FILE_NAME, "r")
34  students = json.load(file_obj)
35  file_obj.close()
36
```

Figure 3: using Python's JSON module to read data from file

The process of writing data to a file, using the JSON module was a reprieve from the code, I was using, that was getting, incrementally, more complex. The task that previously required six lines of code could, now, be achieved using only three lines of code. In this code, I opened the file with "write" permission; used the "json.dump()" to save data to the file and then close the file afterwards. (Figure 4)

```
87          # Save the data to the file
88          file_obj = open(FILE_NAME, "w")
89          json.dump(students, file_obj)
90          file_obj.close()
91          print("Data saved!")
92
```

Figure 4: Saving data to file using the JSON module

Dictionaries

In this week's assignment, I learned how to use "dictionaries" which offered a more intuitive mechanism for referencing values. As an added value, also, it was easier for me to read through my code and quickly identify values and keys, and how and where I used them, compared to the indexing structure that lists offered. (Figure 5)

```
72      elif menu_choice == "2":
73          # Input the data
74          print("-" * 50)
75          student_first_name = input("What is the student's first name? ")
76          student_last_name = input("What is the student's last name? ")
77          student_gpa = float(input("What is the student's GPA? "))
78          # Add student data to a dictionary variable "student-data"
79          student_data = {"FirstName": student_first_name,
80                          "LastName": student_last_name,
81                          "GPA": student_gpa}
82          # Add data to the table
83          students.append(student_data)
84          print("-" * 50)
85          continue
```

Figure 5: key-value pairing simplifying elements of code at first glance

Error Handling

In Assignment03, I included extra code to trap and mitigate unwanted program behavior. In the program definitions, I included a “save_file_counter” of type “int” and initialized it to zero. As I tested my program, I did not like the fact that if the user was able to select the option to “Save to File” multiple times for the same data, and the program would oblige and save the same data to file. So, I coded a flag to catch that and prompt the user that data had already been saved, then increment the flag, and reset it back to zero when user selected choice “1”. In the script I also coded other flags to catch any unwanted behaviors by the program.(Figure 6)

```
28 file_save_counter: int = 0
29
30 # Present and Process the data
31 while True:
32     # Present the menu of choices
33     print(MENU)
34     menu_choice = input("What would you like to do: ")
35     # Input user data, explicitly reset file_save_counter to zero
36     if menu_choice == "1":
37         student_first_name = input("What is the student's first name? ")
38         student_last_name = input("What is the student's last name? ")
39         course_name = input("What is the Course name? ")
40         csv_data = f"{student_first_name},{student_last_name},{course_name}\n"
41         file_save_counter = 0
42
43     # Present the current data
44     elif menu_choice == "2":
45         # check if the string is empty: No data available
46         if csv_data == "":
47             print("No data available at this time!")
48         else:
49             print("The current data is:")
50             print(csv_data)
51     # Save the data to a file
52     elif menu_choice == "3":
53         # if string is empty, no data to save to file, return to menu
54         if csv_data == "":
55             print("No data to record at this time!")
56         # check whether data has already been saved to file #
57         # do not save again - return to menu
58         elif file_save_counter != 0:
59             print("Data has already been saved to file!")
60         # if user has entered data, then save it to file and increment
61         # file_save_counter
62         else:
63             file_obj = open(FILE_NAME, "w")
64             file_obj.write(csv_data)
65             file_obj.close()
66             print("\nData Recorded!")
67             print("Data below was saved to file.")
68             print(csv_data)
69             file_save_counter += 1
70     # Stop the while loop; reset file_save_counter to zero
71     elif menu_choice == "4":
72         file_save_counter = 0
73         print("Program Ended!")
74         break
```

Figure 6: Assignment03.py - attempts to catch unwanted program behavior

While coding the program “Assignment03”, this was the best I could muster — at the time — to control how I wanted my program to behave. Since then, I have learned better ways to catch and mitigate problematic program behavior and inappropriate user input. This week, I learned how to use the structured “Try-Except” construct to catch errors, and minimize the chances of the program crashing.

Below is a snippet of code running the initial file opening and reading of data from the file when the program launches using the “Try-Except” construct. This error handling ensures that the file is indeed available to be read from and not crash the program, but instead displays the appropriate error messages to the user. (Figure 7)

```
33  # When the program starts, read the file data into a list of lists (table)
34  # Extract the data from the file
35  try:
36      file_obj = open(FILE_NAME, "r")
37      students = json.load(file_obj)
38  except FileNotFoundError as e:
39      print("Text file must exist before running this script!\n")
40      print("-- Technical Error Message --")
41      print(e, e.__doc__, type(e), sep='\n')
42  except Exception as e:
43      print("There was a non-specific error!\n")
44      print("-- Technical Error Message --")
45      print(e, e.__doc__, type(e), sep='\n')
46  finally:
47      if file_obj.closed == False:
48          file_obj.close()
49
```

Figure 7: error handling using try-except structure for initial file opening and reading

User input is one of the aspects I was experimenting with in Assignment03. However, I only went as far as checking whether a string was empty, the actual data input by user was not qualified. This week, I also learned how to use the try-except construct to catch errors in user input.

In the code snippet below, I check whether the user did not enter only alphabetic characters for the first and last names, then alerting the user using a custom message, if input did not include only alphabetic characters. Alongside the custom message, technical details are displayed as well.

The visual cues for the levels of indentation in PyCharm made it easier to keep track of the program flow while coding as well as when reviewing the code at a later time. This came in handy when I had to embed the code with the try-except structure.(Figure 8)

```
55
56      # Input user data
57      if menu_choice == "1":  # This will not work if it is an integer!
58          try:  # trap user input errors
59              student_first_name = input("Enter the student's first name: ")
60              if not student_first_name.isalpha():
61                  raise ValueError("First name should only contain alphabetic characters!")
62              student_last_name = input("Enter the student's last name: ")
63              if not student_last_name.isalpha():
64                  raise ValueError("Last name should only contain alphabetic characters!")
65              course_name = input("Please enter the name of the course: ")
66              student_data = {"FirstName": student_first_name,
67                              "LastName": student_last_name,
68                              "Course": course_name}
69              students.append(student_data)
70              print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
71          except ValueError as e:
72              print(e)  # Prints custom message
73              print("-- Technical Error Message --")
74              print(e.__doc__)
75              print(e.__str__())
76          except Exception as e:
77              print("There was a non-specific error!")
78              print("-- Technical Error Message --")
79              print(e, e.__doc__, type(e), sep='\n')
80              print("-" * 50)
81
```

Figure 8: structured error handling for user input using 'try-except' contract

Running and Testing Program in Terminal

After completing the coding and initial program runs and tests in PyCharm, I run the program in Terminal to ensure it run as expected and the error handling worked as expected.

1) The first option I selected on program launch was Option 1: "Register a student for a course". I purposefully entered the an alphanumeric name and the program caught this, displayed a custom message and some technical data.

2)For the second test, I selected Option 3: "Save data to file". The first test did not complete the registration process. Given that there was no new information collected by Option 1, the program displayed a custom message alerting the user that there was no new data to save.

3) When Option 2: “Show current data” was selected, the data saved in the file — that had been read into list table variable “students” at the beginning of the program — was displayed in a custom message.

The screenshot below shows the first three respective test runs and the output in Terminal (Figure 9)

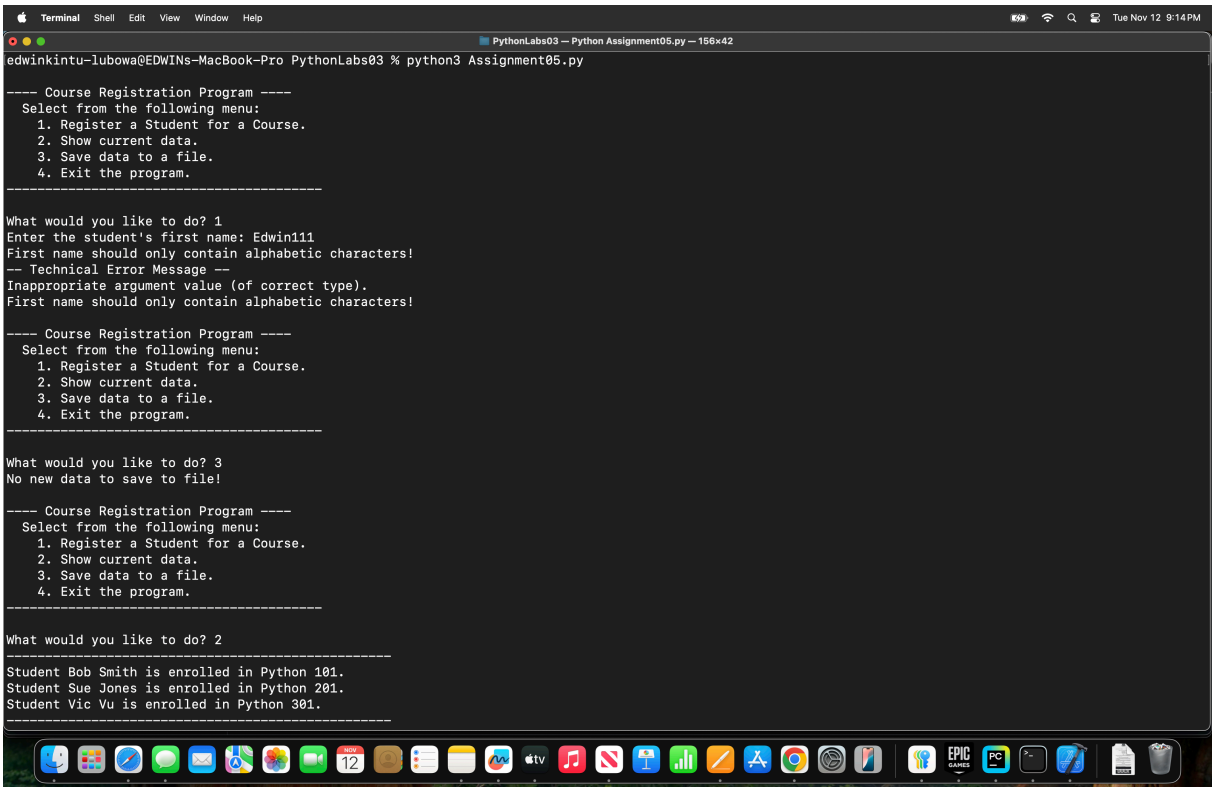

Figure 9: Output in Terminal

The next test, was personal, my last name is hyphenated but some the entities I interact with will not accept the hyphen. There have been many variations of my last name, spawned from the rejection of the hyphen: some have dropped one of the names; others have removed the hyphen and displayed them as two separate names representing a last name; some have removed the hyphen and concatenated the names forming a very long last name. After this week’s class, I’m excited to research and attempt to recreate the ‘solutions after the flag’ that spawned the variants to my last name. Below is a Terminal output rejecting both the hyphen and the space character (Figure 10)

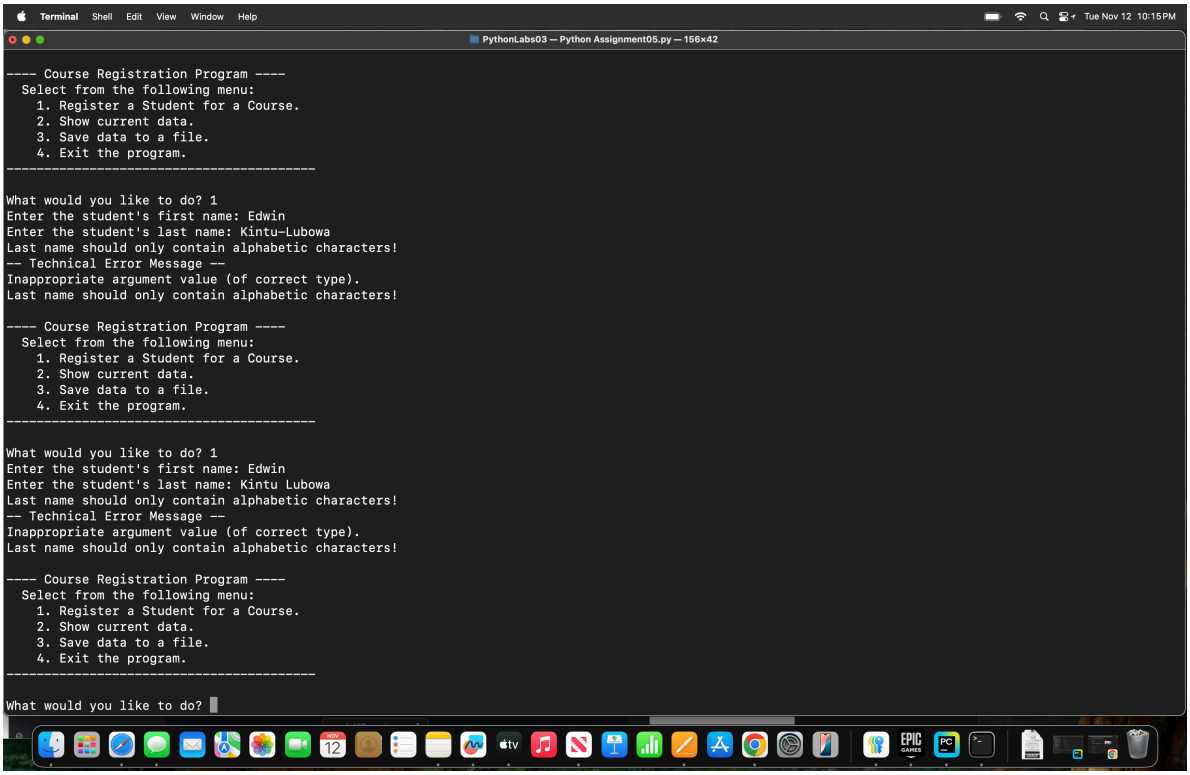

Figure 10: double last names and hyphenated last names causing errors

The Terminal output below shows student registration and saving of the student data. When option 3 is selected, it also displays all the students who have registered thus far. (Figure 11)

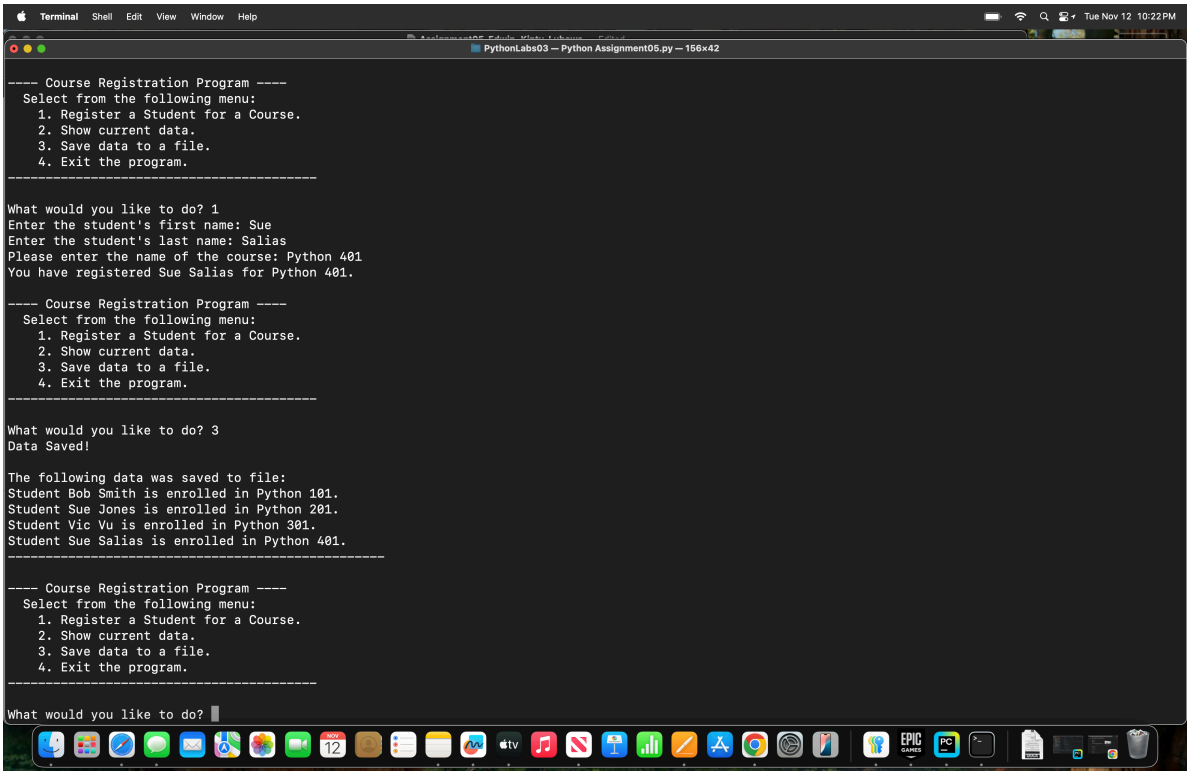

Figure 11: Output showing registration, saving student data to file and displaying all registrants

Summary

The program I wrote this week for the assignment was bigger and more complex than prior assignments. Coding started with a starter file; I updated the header to include my names and date the file was created; Assignment05 was very similar to Assignment04 from last week, however, for this week I coded the first part of the program that read data from a file and saved it to a list table “students”, then run the program to make sure it did not break or crash. I then coded the section to process the data: with the menu having four choices, for each option, I coded, then run the program and this became the common theme for this week’s assignment. Code small sections, run them to ensure they are okay. This is the structure I followed until the successful completion of the program. For as long and as complex the program was, the theme of coding small bits of the program and testing them out before moving on was a great framework for tackling the complexities of the program— that initially, looked ominous.