

Name: Student-PyQ

Date: 02/23/25

Course: IT FDN 110 A Wi 25: Foundations of Programming Python

Assignment: 05

GitHub URL:

<https://github.com/Student-PyQ/IntroToProg-Python-Mod05>

Advanced Data Collection

Introduction

In the last assignment we worked with lists collections. This assignment is a compare and contrast between lists and dictionary collections. In contrast to lists collections, the use of dictionary collections can offer flexibility in more advanced logic for data processing. But unlike the sequential order of lists data elements, some knowledge of the dictionary data structure is necessary to accurately select data values by known key names. With that said, dictionaries can also introduce more errors to data processing, when using inaccurate key names or introducing typos. A structured approach to error handling can elevate some of this burden. The following steps were followed to show how the use of dictionary collections and JavaScript Object Notation (JSON) file types can complement each other.

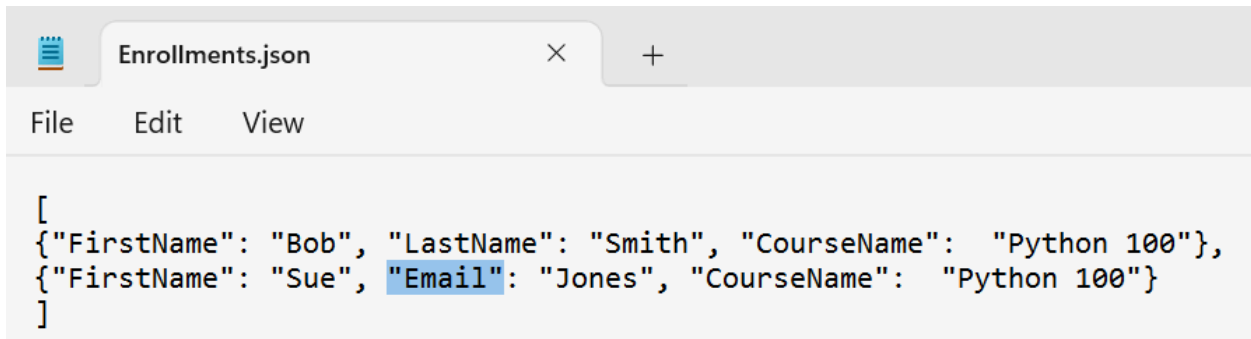
Step 1 - Read from JSON file & add to dictionary

For this step, I started with the starter JSON file and noticed right away that the file had a typo in it which could cause an error (**See Figure 1.1**). I fixed the key name typo by replacing it with the intended key name, LastName. I will add it back later when I incorporate structured error handling and testing in my code. I also used the starter assignment program but added my comments throughout the code where I saw an opportunity to make code edits. Essentially, I added pseudo code in comments to make sure I would not miss any of the assignment acceptance criteria.

Reading from a JSON file was a lot simpler than reading from a csv. Mostly due to using the import JSON library which comes with its own methods like load() and dump() which does all the parsing work for you. For step 1, I needed to read from the JSON file using the dump method (**See Figure 1.2**). The benefits of using the dump() method is that it converts the file contents directly into a list of dictionary collection types. Which saves a lot of lines of code in the program.

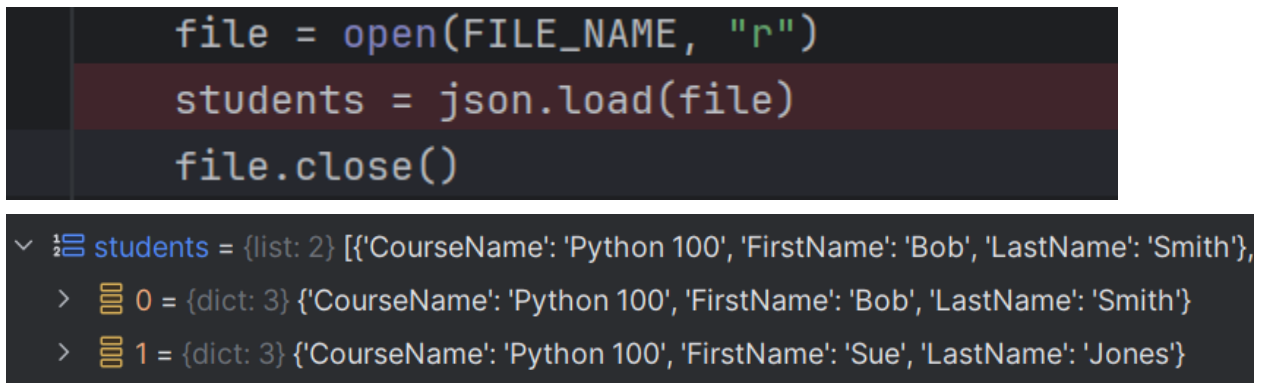
The next area of the existing starter program that I needed to modify is when selecting menu option 1, Register a Student for a Course. I would need to add new student details to the list of dictionaries recently loaded from a JSON file. I found this step simple in that I just had to replace the existing code to reference the student collection by dictionary key, instead of a list index value (**See Figure 1.3**). I was also able to keep the rest of the code intact because the `student_data` variable was changed from a list collection to a dictionary collection type. And `students` variable is simply a list where the append method would still be appropriate. I also used the same approach for modifying menu option 2, Show current data.

Figure 1.1 - Enrollments.json file



```
[
{"FirstName": "Bob", "LastName": "Smith", "CourseName": "Python 100"},
{"FirstName": "Sue", "Email": "Jones", "CourseName": "Python 100"}
]
```

Figure 1.2 - JSON.load method



```
file = open(FILE_NAME, "r")
students = json.load(file)
file.close()
```

```
> students = {'list': 2} [{'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'},
> 0 = {'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}
> 1 = {'CourseName': 'Python 100', 'FirstName': 'Sue', 'LastName': 'Jones'}]
```

Figure 1.3 - Reference by Key for dictionary value

```
# Input user data
if menu_choice == "1": # This will not work if it is an integer!
    student_first_name = input("Enter the student's first name: ")
    student_last_name = input("Enter the student's last name: ")
    course_name = input("Please enter the name of the course: ")
    # 2/23 Change: Add entered student data to dictionary collection.
    student_data = {"FirstName": student_first_name, "LastName": student_last_name, "CourseName": course_name}
    students.append(student_data)
    print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
    continue

# Present the current data
elif menu_choice == "2":

    # Process the data to create and display a custom message
    print("-"*50)
    for student in students:
        # 2/23 Change: Modified to get value by dictionary key.
        print(f"Student {student["FirstName"]} {student["LastName"]} is enrolled in {student["CourseName"]}")
    print("-"*50)
    continue
```

Step 2 - Save to JSON file & print from dictionary

In step 2, I used the dump method of the JSON library (**See Figure 2.1**). In comparison to the use of list collection with csv file for storage, the dictionary collection structure offers an intuitive approach to saving data and uses less code. Rather than needing to loop through a list of student data to string format data for saving to a csv file. I only needed to pass in the dictionary list collection and file object variables when using the JSON dump method to save the data to a JSON file. However when using the print method, I still had to loop through the collection list to display registered students that were saved to file (**See Figure 2.2**). I learned that if convenience in code writing and readability is desired then dictionaries and JSON files are the way to go for smaller data processing tasks.

Figure 2.1 - JSON.dump method

```
# Save the data to a file
elif menu_choice == "3":
    # 2/23 Change: Save to json file using the dump method.
    """file = open(FILE_NAME, "w")
    for student in students:
        csv_data = f"{student[0]},{student[1]},{student[2]}\n"
        file.write(csv_data)
    file.close()"""
    file = open(FILE_NAME, "w")
    json.dump(students, file)
    file.close()
```

Figure 2.2 - Access dictionary item value by key

```
print("The following data was saved to file!")
for student in students:
    # 2/23 Change: Modified to get value by dictionary key.
    print(f"Student {student['FirstName']} {student['LastName']} is enrolled in {student['CourseName']}")
    continue
```

Step 3 - Use Structured Error Handling

In the last assignment, I added some error checking by using if and else statements. But in this assignment I learned about the benefits of using structured error handling, Try-Except-Finally blocks. Try-Except blocks are useful because most of the coding for handling or responding to errors gracefully is already done. And the option to customize error messages is built into the logic. The first area where I discovered error handling would be useful was in step 1, when I identified a dictionary key typo in the Enrollments file. To test the Try-Except block, I used examples from the course lab and added back in the key typo in the Enrollments.json file (**See Figure 3.1**). After debugging the code, I realized that the error would not surface until attempting to access a specific key name that did not exist in the json file for a specific record (**See Figure 3.2**). In this case, the code was looking for 'LastName' but instead 'Email' was entered in the Enrollment.json file by error, which caused a key error at runtime. Therefore, I adjusted my code by adding a Try-Except block around the code accessing the dictionary data value by key name and including a customized error message (**See Figure 3.3**). I added the same structured error handling in menu option 3, Save data to a file.

Figure 3.1 - Try-Except-Finally block & Dictionary key typo in file

```
42
43 # When the program starts, read the file data into a list of lists (table)
44 # Extract the data from the file
45 # 2/23 Change: Changed to read load from Json file, add try-except block.
46 try:
47     file = open(FILE_NAME, "r")
48     students = json.load(file)
49     file.close()
50 except FileNotFoundError as e:
51     print("Text file must exist before running this script!\n")
52     print("-- Technical Error Message -- ")
53     print(e, e.__doc__, type(e), sep='\n')
54 except Exception as e:
55     print("There was a non-specific error!\n")
56     print("-- Technical Error Message -- ")
57     print(e, e.__doc__, type(e), sep='\n')
58 finally:
59     if file.closed == False:
60         file.close()
```

Enrollments.json ×

```
"LastName": "Smith", "CourseName": "Python 100", {"FirstName": "Sue", "Email": "Jones", "CourseName": "Python 100"}]
```

Figure 3.2 - Key error 'LastName'

```
print(f"Student {student["FirstName"]} {student["LastName"]} is enrolled in {student["CourseName"]}")
~~~~~^~~~~~
```

KeyError: 'LastName'

```
# Present the current data
elif menu_choice == "2":

    # Process the data to create and display a custom message
    print("-"*50)
    for student in students:
        # 2/23 Change: Modified to get value by dictionary key.
        print(f"Student {student["FirstName"]} {student["LastName"]} is enrolled in {student["CourseName"]}")
    print("-"*50)
    continue
```

"LastName" key is not found because "Email" is the key name instead, as a typo in the Enrollments.json file.

Figure 3.3 - Try-Except

```
elif menu_choice == "2":

    # Process the data to create and display a custom message
    print("-"*50)
    for student in students:
        # 2/23 Change: Modified to get value by dictionary key.
        try:
            print(f"Student {student['FirstName']} {student['LastName']} is enrolled in {student['CourseName']}")
        except Exception as e:
            print("There was a error with the student data!\n")
            print("-- Technical Error Message -- ")
            print(e, e.__doc__, type(e), sep='\n')
    print("-"*50)
    continue
```

Summary

In summary, dictionary data collections can offer more flexibility and nesting data structures for complex data processing. Although it can be more prone to data processing errors if the data structure is not known or has errors. But these disadvantages can be compensated by using the JSON library and incorporating some structured error handling.

Console Test Screen Prints:

```
Command Prompt - python / X + v

3. Save data to a file.
4. Exit the program.
-----

What would you like to do: 2
-----

Student Bob Smith is enrolled in Python 100
Student Sue Jones is enrolled in Python 100
-----

---- Course Registration Program ----
Select from the following menu:
1. Register a Student for a Course.
2. Show current data.
3. Save data to a file.
4. Exit the program.
-----

What would you like to do: 1
Enter the student's first name: James
Enter the student's last name: Brown
Please enter the name of the course: Python 500
You have registered James Brown for Python 500.

---- Course Registration Program ----
Select from the following menu:
1. Register a Student for a Course.
2. Show current data.
3. Save data to a file.
4. Exit the program.
-----

What would you like to do: |
```

```
---- Course Registration Program ----
Select from the following menu:
    1. Register a Student for a Course.
    2. Show current data.
    3. Save data to a file.
    4. Exit the program.
-----

What would you like to do: 3
The following data was saved to file!
Student Bob Smith is enrolled in Python 100
Student Sue Jones is enrolled in Python 100
Student James Brown is enrolled in Python 500

---- Course Registration Program ----
Select from the following menu:
    1. Register a Student for a Course.
    2. Show current data.
    3. Save data to a file.
    4. Exit the program.
-----

What would you like to do: 4
Program Ended
```



The screenshot shows a code editor window with a single tab titled 'Enrollments.json'. The editor has a menu bar with 'File', 'Edit', and 'View' options. The JSON content is as follows:

```
[{"FirstName": "Bob", "LastName": "Smith", "CourseName": "Python 100"}, {"FirstName": "Sue", "LastName": "Jones", "CourseName": "Python 100"}, {"FirstName": "James", "LastName": "Brown", "CourseName": "Python 500"}]
```

The third object in the array is highlighted with a blue selection background.

References

Root, R. (2025). Module 05 - Advanced Collections and Error Handling. In IT FDN 110 A Winter 2025, *Introduction to Programming with Python*, (pp. 1-35) University of Washington