Derek Larson

11/12/2023

IT FDN 110: Foundations of Programming-Python

Assignment 05

https://github.com/derkrylar99/IntroToProg-Python/blob/main/Assignment05.py

Dictionaries, JSON, and Error Handling

Introduction

Introducing the flexible hierarchal structure of Dictionaries and JSON files to the "Student Registration Program", this module introduces the use of Dictionaries embedded in Lists as well as how to read/write/append data from JSON files. Importing the JSON module allows for easy import and export of data, with both Python and JSON using similar data formatting with Lists of Dictionaries. Data is loaded and dumped to the file using JSON module which greatly reduces code lines. Structured error handling is integrated to provide more readable, specific error messages for anticipated user mistakes or uncontrollable bugs.

Creating the Program

Reviewed the example "Assignment05-Starter.py", opened the previous "Assignment04.py" file submitted for Module 4, and modified the script to become "Assingment05.py":

- Saved as new file "Assignment05.py" and updated Header comments information.
- Streamlined code using reference file as comparison:
    - Removed checks for empty csv_data variable (checking == '')
    - Removed exit() and utilized break for menu_choice == "4", to officially break the 'while' loop
    - Moved the exiting print statement into the while loop indent
    - Removed or comment-out any references to csv_data variable (method replaced with dictionaries)

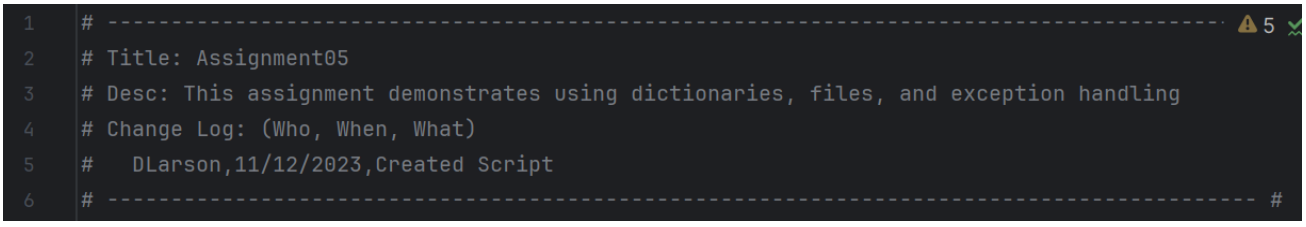

Figure 1.1: Header information updated when Assignment04 is edited to become the basis for Assignment05.

Importing Modules

To utilize Python's built-in JSON module, it must be imported at the script start (just underneath the Header comments). This will allow for streamlining in code later on, with significant reduction in lines when using custom JSON methods.

```
7   # Import code from Python's JSON module
8   import json
9
```

Figure 1.2: Importing the JSON module in the Python script.

Establishing the Variables

The Constant variables, which will not be changed throughout the execution of the program, are the first set of variables to be established. Each variable is created using the convention for Constants (ALL CAPS) and assigned to the String type.

```
10  # Define the Data Constants
11  MENU: str = '''
12  ---- Course Registration Program ----
13    Select from the following menu:
14      1. Register a Student for a Course
15      2. Show current data
16      3. Save data to a file
17      4. Exit the program
18  ----------------------------------------
19  '''
20  FILE_NAME: str = "Enrollments.json"
21
```

Figure 1.3: Assigning the Constant variables and JSON file format.

- FILE_NAME variable reset to "Enrollments.json" file type, to replace the .csv format
- MENU variable contains the primary display menu to prompt the user for interaction. Since the displayed menu of options contains many lines and some complex formatting, it can by typed exactly as it will appear when displayed on the screen within the "'<>''' multi-line comment.

Next, we define all data variables by type, before they are assigned later during the program's execution. Four variables are needed to store user input and are defined as student_first_name, student_last_name, course_name, and menu_choice. The csv_data variable is no longer used, but still exists within the comment-disabled lines, so it will be retained in the definition list. The "file" variable will be set to None;

it will not be assigned the Object type hint due to issues with PyCharm. The menu_choice variable holds the user's menu selection.

The students variable will remain as "list" type, and continue to act as the "List of lists"; however, the student_data variable is set to Dictionary type using curly brackets instead of flat brackets: "dict = {}" and replaces the List type to become a "List of Dictionaries". This will allow for searching or setting variables using named Key values instead of reliance on index positioning.

```
22  # Define the Data Variables
23  student_first_name: str = ''
24  student_last_name: str = ''
25  course_name: str = ''
26  csv_data: str = ''
27  file = None
28  menu_choice: str = ''
29  student_data: dict = {}
30  students: list = []
```

Figure 1.4: Defining the data variables for assignment, with type associations.

Reading the JSON File Data and Loading into Lists

Because the JSON module was imported at the beginning of the script, the custom built-ins can be used to load the contents of the JSON file and convert it to a Python List of Dictionaries. Instead of splitting rows and converting data into Lists, the file can be loaded and the contents assigned directly to the students variable – immediately recognizable to Python as a List of Dictionaries due to the formatting style of the JSON data structure:

- Open JSON file in read mode with "r" : open(FILE_NAME, "r")
- Assign the contents of the JSON file to the students variable:
    - students = json.load(file)
- file is closed and print statement for status

This is the first implementation of Structured Error Handling, where the JSON file load is embedded into a "try except" loop. First it will "try" to open and read the file specified, load the contents into the students variable, and IF it fails the code will fall back to three specified exceptions with custom message printouts.

- except TypeError as e: to specify if the user provides the wrong type of file
    - prints custom message to check file for JSON format
    - prints a 'heads-up' message to indicate more technical error reporting will follow
    - prints the error Exception object encountered (retained by Python) as e
        - prints documentation string of exception type "e.__doc__)"
        - prints the type of exception object "type(e)"
        - separate each printout with a new line: sep='\n'
- except FileNotFoundError as e: to specify errors related to the file not being found/not existing

    - repeat same print out statements as above:
        - custom message to inform user the file must exist
        - technical messages following
        - exception object, documentation string, exception object type, separated by new lines
- except Exception as e: to catch any other errors encountered during the file read process
    - custom message the user has encountered a non-specific error
    - repeat same print out statements as above:
        - custom message to inform user the file must exist
        - technical messages following
        - exception object, documentation string, exception object type, separated by new lines
- finally: the last specification to catch any cases where the file has remained in an open state in memory
    - will check whether the file is still open with a Boolean: if file.closed == False:
    - triggers the file to be closed: file.close()

```
# Extract using JSON instead of CSV
try:
    file = open(FILE_NAME, "r")
    students = json.load(file)
    file.close()
    print("Data has been processed!")
# Provide structured error handling
except TypeError as e:
    print("Please check the data is a valid JSON format\n")
    print("-- Technical Error Message -- ")
    print(e, e.__doc__, type(e), sep='\n')
except FileNotFoundError as e:
    print("Text file must exist before running this script!\n")
    print("-- Technical Error Message -- ")
    print(e, e.__doc__, type(e), sep='\n')
except Exception as e:
    print("There was a non-specific error!\n")
    print("-- Technical Error Message -- ")
    print(e, e.__doc__, type(e), sep='\n')
finally:
    if file.closed == False:
        file.close()
```

Figure 1.5: Attempting to load a JSON file, and customized "user friendly" error reporting for most anticipated scenarios (if file loading fails).

Collecting and Formatting the Data

The primary structure of the user-selected options is still embedded within the: while True:

- prints the Constant variable MENU set with custom message to display
- prompts the user for input and assigns it to the menu_choice variable (to be read and matched to an if/elif option)

```
66  # Present and Process the data
    while True:
68      # Present the menu of choices
69      print(MENU)
70      menu_choice = input("Please enter your Menu choice number: ")
71
```

Figure 1.6: Main menu display and input prompts at the top of the "while" loop.

Menu Choice 1 : Input User Data

To input and store user data, the variables for student_first_name, student_last_name, and course_name are assigned using the input and custom prompt message. This is another implementation of Structured Error Handling, to provide more understandable reporting when numbers or symbols are accidentally entered into the First or Last Name entries.

- User is prompted with a message to enter the First and Last Name
- Check for any non-letter characters: if not student_first_name.isalpha() where the .isalpha() checks for invalid character entries in the string
- Raises a customized error message using the ValueError with pre-typed display message
- (^ steps repeated for the Last Name)

After taking in the First and Last names, the user is prompted to input the course name which is assigned to variable course_name. The three user input variables are assigned to the student_data dictionary with matching Key values, defined before appending the student_data dictionary into the students List.

- student_data = {'FirstName': student_first_name, 'LastName': student_last_name, 'CourseName': course_name}
- append the newly-entered student information (converted to dictionary entry within the student_data variable) to the students List
    - students.append(student_data)
- end with a print statement using the entered variables, displayed with a custom formatted message using the f-string method

Finally, there is the remaining implementation of the Structured Error Handling for First Name + Last Name user input. Exactly as with the file verification/error handling, each except provides instructions for predicted possible failures:

- ValueError
    - custom message the user has encountered a Value Error:

        - custom message to inform user of more technical information
        - print: exception object, documentation string, exception object type, separated by new lines
- Exception as e: - catch-all for other errors
    - Custom message for 'non-specific error'
    - Message for more technical information
        - print: exception object, documentation string, exception object type, separated by new lines

```
    # Input user data
    if menu_choice == "1":
        # Structured error handling for non-letter characters
        try:
            student_first_name = input("Please enter the student's First Name: ")
            if not student_first_name.isalpha():
                raise ValueError("The First Name should only contain letters!")

            student_last_name = input("Please enter the student's Last Name: ")
            if not student_last_name.isalpha():
                raise ValueError("The Last Name should only contain letters!")

            course_name = input("Please enter the Course Name: ")
            # Add the student data to a Dictionary matching students table keys
            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}.")
        except ValueError as e:
            print(e) # Prints custom message
            print("-- Technical Error Message -- ")
            print(e.__doc__)
            print(e.__str__())
        except Exception as e:
            print("There was a non-specific error!\n")
            print("-- Technical Error Message -- ")
            print(e, e.__doc__, type(e), sep='\n')
        continue
```

Figure 1.7: User data is captured and added to the "students" List, with Customized Error Handling

Menu Choice 2 : Present the Data

If the user enters Menu choice "2", the menu_choice variable will trigger the elif statement for "Presenting the current data" (as indicated in the comment line above elif menu_choice == "2":).

To print all the data, both entered by the user and read from the JSON file, the students List is cycled through using a for-loop. For each entry "student" in students, a print statement is formatted (using the f-string method) inserting the dictionary elements into the proper slots using the matching Key value (this is an improvement over the List method, where index numbers must be used creating a reliance on

strict index ordering. Instead, with Dictionaries the Key values can be used instead of index numbers which creates more flexibility).

This printed list is embedded within two rows of asterisks, using the print("*"*50) to multiply the "*" character by 50 times, for improved visibility on the screen.

```
102     # Present the current data
103     elif menu_choice == "2":
104         print("-"*50)
105         for student in students:
106             print(f"Student {student['FirstName']} {student['LastName']} is enrolled in {student['CourseName']}")
107         print("-"*50)
108         continue
109
```

Figure 1.8: Loading the contents of the file into the empty "csv_data" using f-string formatting.

Menu Choice 3 : Save the Data to the File

By selecting the option to "Save data to a file", the user will enter 3 and match to the condition in the second elif statement (specified in Python with elif menu_choice == "3": for when the menu_choice variable is equal to "3").

The code used to open/save the .csv file (plus formatting for the csv data input) has been commented out for this code. The unused lines are the only place using the deprecated csv_data variable.

```
110     # Save the data to a file
111     elif menu_choice == "3":
112         # file = open(FILE_NAME, "w")
113         # for student in students:
114         #     csv_data = f"{student[0]},{student[1]},{student[2]}\n"
115         #     file.write(csv_data)
116         # file.close()
117         #
```

Figure 1.9: Deprecated code for handling CSV file prep/writing, commented out for JSON implementation.

Instead of CSV, the program will utilize code to process JSON files and save data directly to them without additional formatting. It will do this nested within another "try except" loop, to provide more Structured Error Handling.

- First it will "try" to open the file (specified in the Constant variables FILE_NAME) in write mode
    - File = open(FILE_NAME, "w")
- Because the JSON module was imported at the beginning of the script, it will save the contents of the students List to the JSON using the imported method, allowing for the simplified code to specify the students variable, file variable, and json.dump built-in function.
- Next it will print the contents of the students List to the screen by cycling through each entry and printing a custom-formatted message using a for-loop (as before, in Option 2 elif):

- For each entry “student” in students, a print statement is formatted (using the f-string method) inserting the dictionary elements into the proper slots using the matching Key Value (rather than index position).
- This printed list is embedded within two rows of asterisks, using the print(“*”*50) to multiply the “*” character by 50 times, for improved visibility on the screen.

- Similar to when the file is read and the students List is initially populated, the same list of checks is run for improved error handling and reporting to the user
  - Except TypeError as e:
    - Custom message for file type being incorrect, check to ensure is JSON
    - More technical info to follow
    - print: exception object, documentation string, exception object type, separated by new lines
  - except Exception as e:
    - custom message for technical info to follow
    - built-in python error reporting information message
    - printing the actual Python error information
      - print: exception object, documentation string, exception object type, separated by new lines
  - finally: will check if the file is already open using a Boolean (if the condition ‘the file is closed’ equals False, then the command to close the file will be executed

```
118         # Using JSON instead of CSV
119         try:
120             file = open(FILE_NAME, "w")
121             json.dump(students, file)
122             file.close()
123             print("*" * 50, "\n")
124             print(f"Your data has been saved in {FILE_NAME}!\n")
125             for student in students:
126                 print(f"Student {student['FirstName']} {student['LastName']} is enrolled in {student['CourseName']}")
127             continue
128         except TypeError as e:
129             print("Please check the data is a valid JSON format\n")
130             print("-- Technical Error Message -- ")
131             print(e, e.__doc__, type(e), sep='\n')
132         except Exception as e:
133             print("-- Technical Error Message -- ")
134             print("Built-In Python error info: ")
135             print(e, e.__doc__, type(e), sep='\n')
136         finally:
137             if file.closed == False:
138                 file.close()
```

Figure 1.10: Attempting to open/write/save/close the JSON file, otherwise report the error with customized formatting for readability.

Menu Choice 4 : Exiting the Program

When the user choses to exit the program, the entry “4” will be read from the menu_choice variable and match the conditions for the final elif statement (elif menu_choice == “4”:). This will print a message to the screen informing the user of the selection to exit, and then close the process using the exit() function.

```
140         # Stop the loop
141         elif menu_choice == "4":
142             break
```

Figure 1.11: Breaking the loop using “break” before printing a status update before exiting.

The “else” Catch-all

To account for accidental keystrokes or misinterpretation, the else statement accounts for any other string assigned to the menu_choice variable and prints a display message on screen to select a valid option. It uses the continue to automatically loop the user back to the Main Menu.

When the loop is ended with the “break”, a print statement will inform the user of the selection to close the program before it terminates.

```
143     else:
144         # For any non-valid selection entered into menu_choice:
145         print("INVALID SELECTION! Please select  1 - 4 from the Menu options.")
146         continue
147 
148 print("You have selected to Close the Program.")
149 
```

Figure 1.12: All other characters assigned to the menu_choice variable go here.

Testing the Program

Now that the code runs properly when testing within PyCharm, it needs to be verified as functional outside of the IDE. To achieve this, the script is run in command shell by navigating to the directory where the file is stored and using the “python” command followed by the file name “Assignment05.py”.

- tested and confirmed the following:
    - error handling when the file is read into the list of dictionary rows
        - deleted the file from the directory, to mimic the file not existing
    - error handling for First and Last name
        - detects the presence of number or symbol characters
        - reports error to inform user to enter only letters

    - error handling for when dictionary rows are written to the file
        - to test: changed FILE_NAME constant to use .csv instead of .json
        - invalid file format error triggered
    - user can input student information: first name, last name, course name
        - input is saved to a dictionary row and assigned the proper key values
        - dictionary rows are appended to the students List of dictionaries
    - user can input multiple student registrations
    - user can display and save multiple student registrations
    - program runs correctly in IDE and console

Summary

In this module, the continuously improving "Student Registration Program" added the ability to structure and access data in even more complex (and user-friendly) ways:

- Building on simple csv structure and List functionality with introduction to JSON file format and Dictionary functionality.
    - Flexible hierarchical format allows for complex data structures to be represented
    - Very human-readable and
    - Is not ideal for large datasets needing to be bulk processed
- Implementing structured error handling to minimize confusion and help guide user to successful program use
- Using git-hub repository to upload and share files