Derek Larson

11/19/2023

IT FDN 110: Foundations of Programming-Python

Assignment 06

github.com/derkrylar99/IntroToProg-Python

Classes, Functions, Separation of Concerns

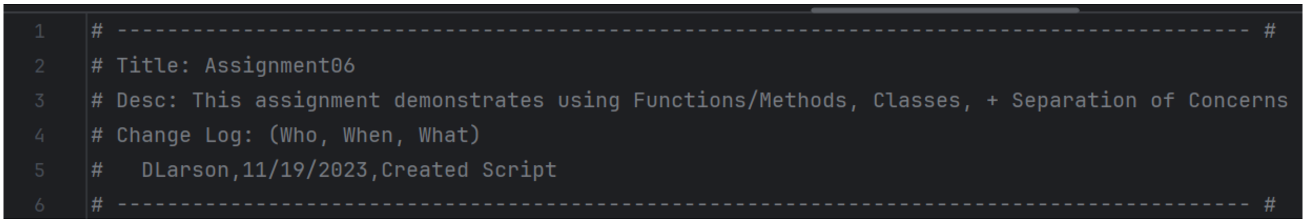
Introduction

In Module 06 the concept of Functions/Methods and Classes reveals how code can become organized within hierarchal groups, with variables / parameters / arguments allowing for data flow between levels of access. Rather than repeat the same instructions in various places, code can be contained within reusable blocks for easy reuse and modification. Care must be taken to avoid “side effects” from overuse of variables which can be unintentionally altered by code from unexpected places.

For this iteration of the program most of the new code provides organization with the introduction of classes and functions, whereas the bulk of the actual script will simply be existing code moved into the appropriate newly defined sections.

Creating the Program

- Opened and reviewed starter file “Assignment06-Starter.py”
    - Diff’d against previous submission “Assignment05.py”
    - Saved previous submission “Assignment05.py” as new file “Assignment06.py”
    - Altered “Assignment06.py” to match functional elements with the starter file
        - Maintained original code, adjusted only where it could affect functionality for this assignment
    - Updated Comment Header script

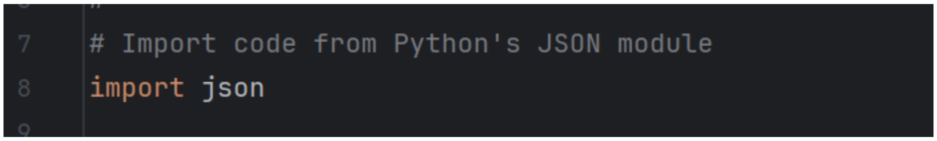


```
# ------------------------------------------------------------------------------------------ #
# Title: Assignment06
# Desc: This assignment demonstrates using Functions/Methods, Classes, + Separation of Concerns
# Change Log: (Who, When, What)
#   DLarson,11/19/2023,Created Script
# ------------------------------------------------------------------------------------------ #
```

Figure 1.1: Header information updated when Assignment05 is edited to become the basis for Assignment06.

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 code later, with significant reduction in lines when using custom JSON methods.

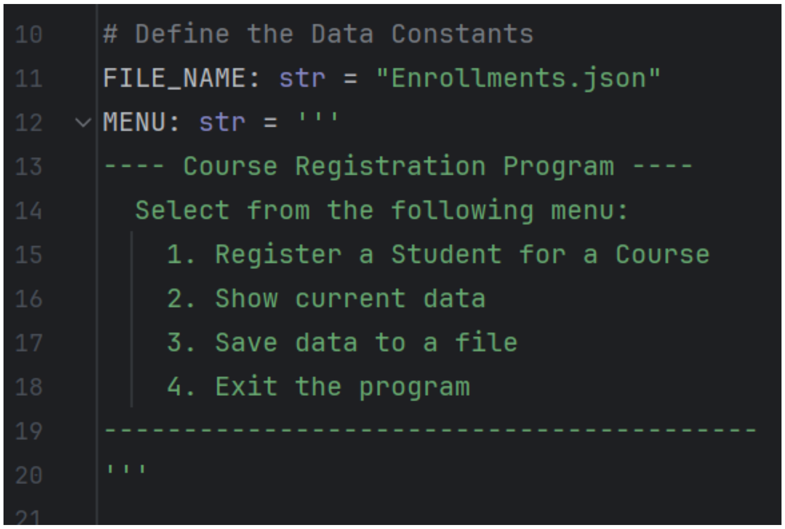


```python
# Import code from Python's JSON module
import json
```

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.



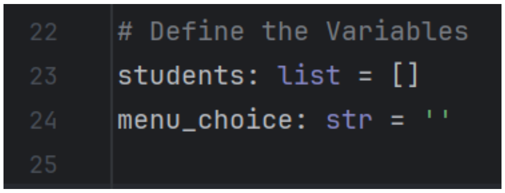
```python
# Define the Data Constants
FILE_NAME: str = "Enrollments.json"
MENU: str = '''
---- Course Registration Program ----
  Select from the following menu:
    1. Register a Student for a Course
    2. Show current data
    3. Save data to a file
    4. Exit the program
-----------------------------------------
'''
```

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

- FILE_NAME reset to "Enrollments.json" file type
- MENU 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 the Data Variables on the global level; in the past all variables were defined here, however with the introduction of Classes and Functions/Methods this list is shortened with most variables nested within the scope of their functions (for predictability, flexibility, and encapsulation).

For the remaining Global variables, the "students" list is defined as empty using an empty bracket set (students: list = []) and the "menu_choice" string with empty single-quotes (menu_choice: str = '').



```
# Define the Variables
students: list = []
menu_choice: str = ''

```

Figure 1.4: Defining the global data variables, with most variables contained within Functions.

Processing the Data: Class “FileProcessor”

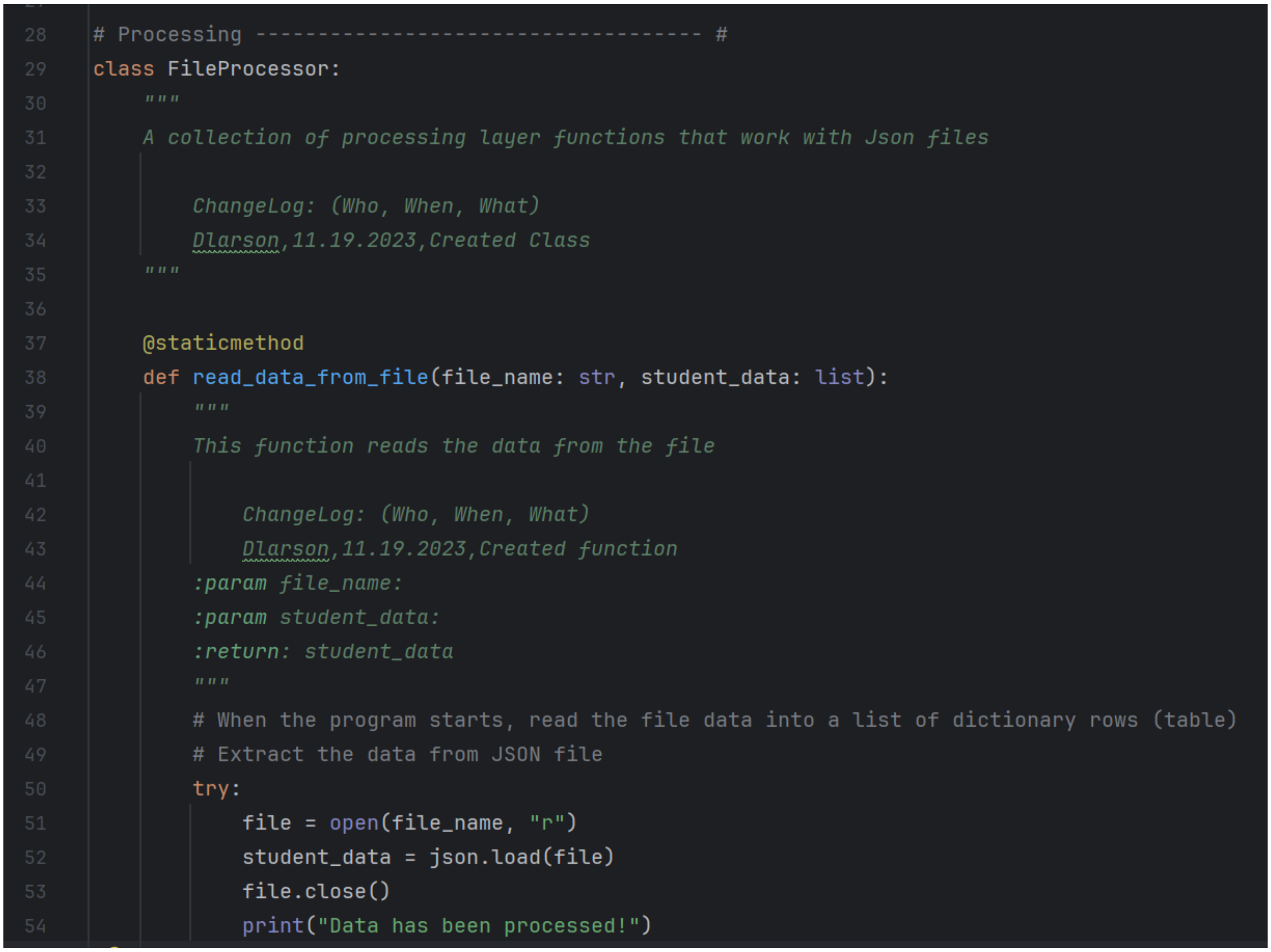
Along with the introduction of Classes and Functions/Methods, the Separation of Concerns allows each portion of the program to be categorized and each specific purpose isolated for maximum modularity and efficiency.

The next section is the “Processing” section marked with the comment line. Within the Processing section are the FileProcessor class and related functions.

Function: read_data_from_file

After the descriptive docstrings, the first function “read_data_from_file” is established with input parameters file_name (a string) and student_data (a list).

- Designated with @staticmethod indicating the function will NOT be changed
    - Allows it to be called without creating an instance object first.
    - **EVERY** function created for this assignment will us the @staticmethod decorator.
- Docstring with function details
- Most of the code here is reused but moved from the global scope into the function scope
    - Indenting is critical
    - 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

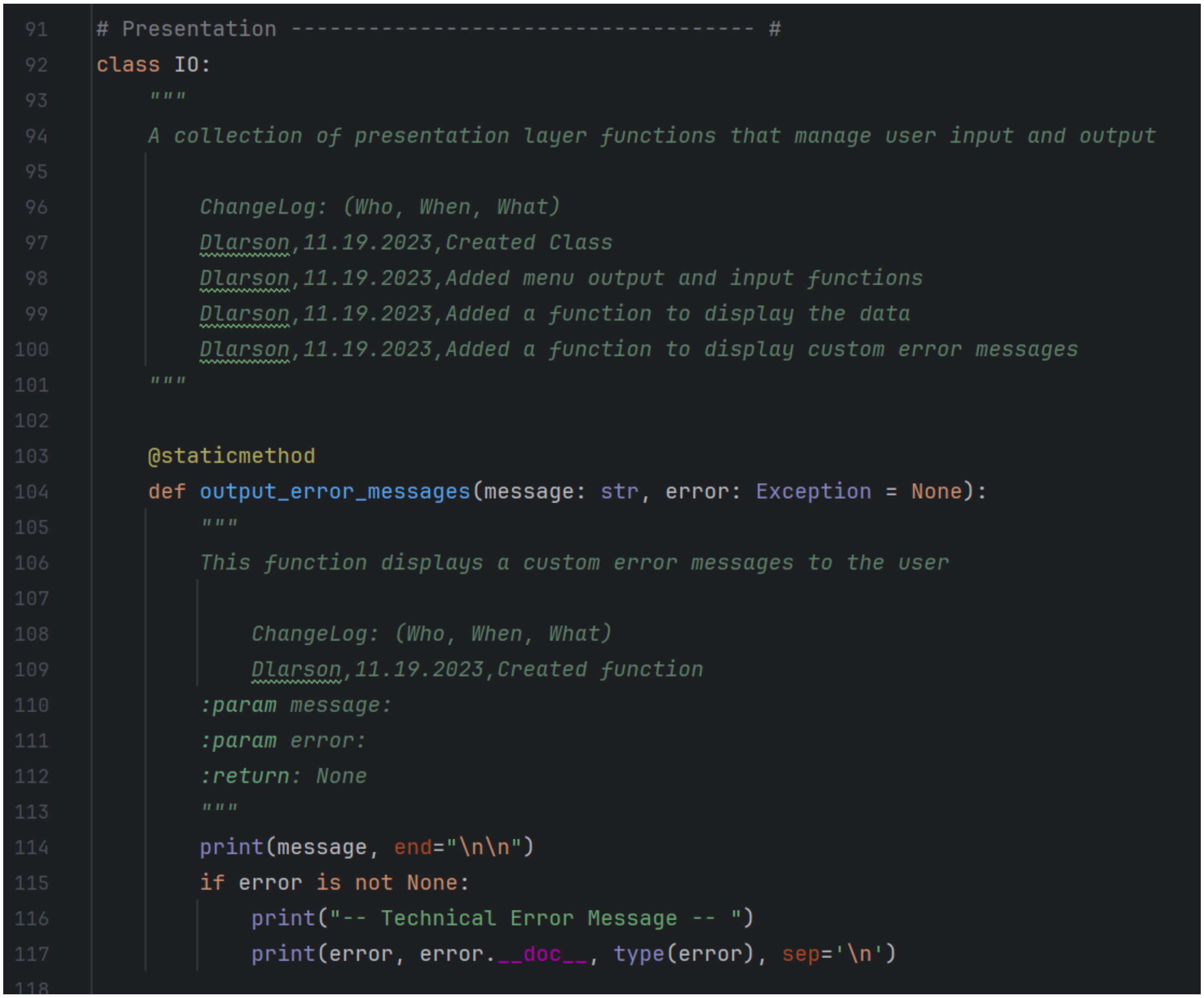


```
# Processing ------------------------------------ #
class FileProcessor:
    """
    A collection of processing layer functions that work with Json files

        ChangeLog: (Who, When, What)
        Dlarson,11.19.2023,Created Class
    """

    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """
        This function reads the data from the file

            ChangeLog: (Who, When, What)
            Dlarson,11.19.2023,Created function
        :param file_name:
        :param student_data:
        :return: student_data
        """
        # When the program starts, read the file data into a list of dictionary rows (table)
        # Extract the data from JSON file
        try:
            file = open(file_name, "r")
            student_data = json.load(file)
            file.close()
            print("Data has been processed!")
```

Figure 1.5: The new FileProcessor Class and file reader Function

- The error handling code is inserted repeatedly throughout the program and should be consolidated into a single function to reuse.

A new Class and Function are created within the “Presentation” section (marked with a comment line after the “Processing” section is finished), to handle the error handling and information formatting. This is created now so it can be immediately implemented within the “read_data_from_file” function.



```python
# Presentation ------------------------------------ #
class IO:
    """
    A collection of presentation layer functions that manage user input and output

        ChangeLog: (Who, When, What)
        Dlarson,11.19.2023,Created Class
        Dlarson,11.19.2023,Added menu output and input functions
        Dlarson,11.19.2023,Added a function to display the data
        Dlarson,11.19.2023,Added a function to display custom error messages
    """

    @staticmethod
    def output_error_messages(message: str, error: Exception = None):
        """
        This function displays a custom error messages to the user

            ChangeLog: (Who, When, What)
            Dlarson,11.19.2023,Created function
        :param message:
        :param error:
        :return: None
        """
        print(message, end="\n\n")
        if error is not None:
            print("-- Technical Error Message -- ")
            print(error, error.__doc__, type(error), sep='\n')
```

Figure 1.6: Input-Output Class with Docstring, followed by the Error Message Output Function

*Function: output_error_messages

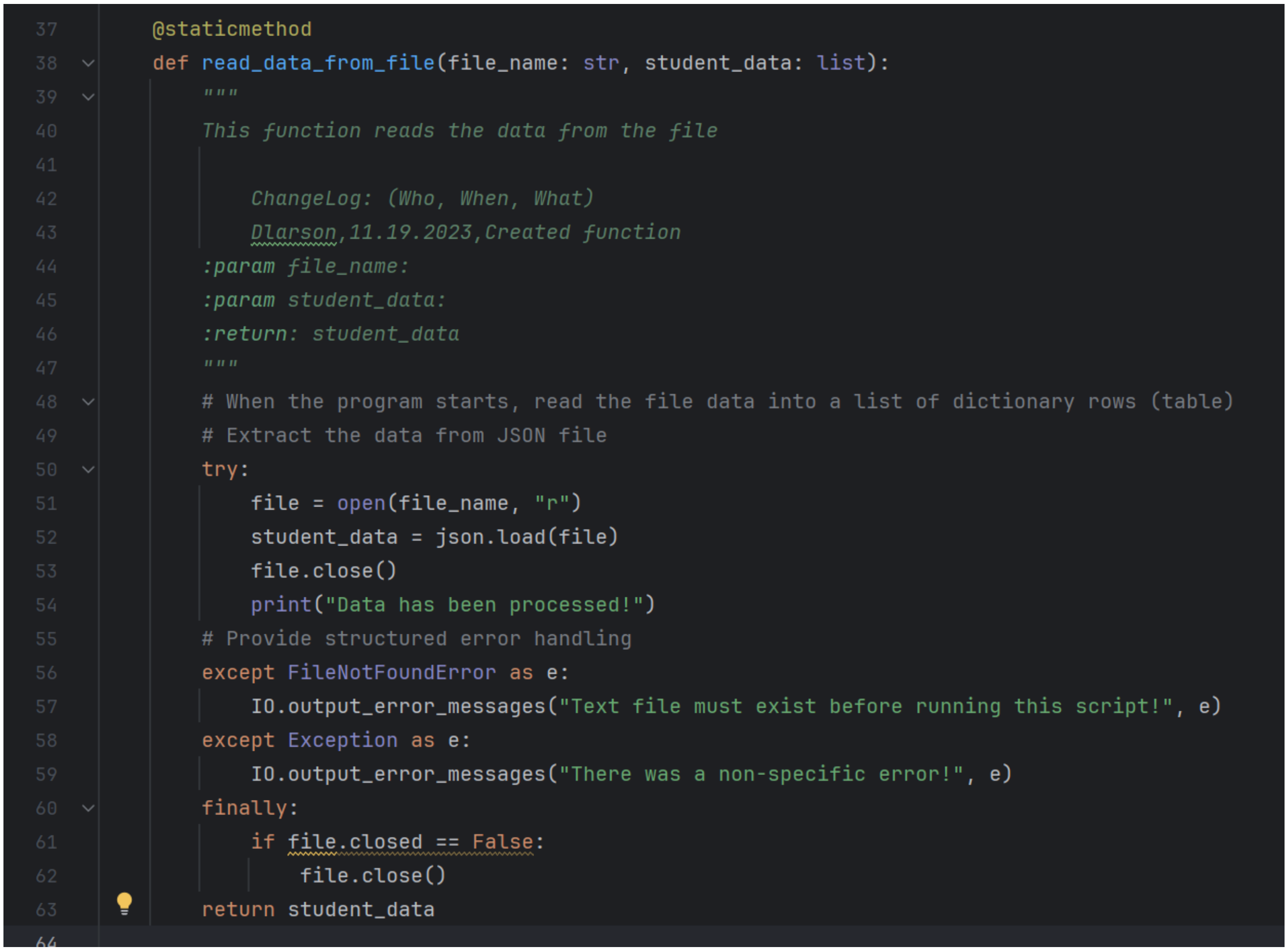
The new function "output_error_messages" (using the @staticmethod decorator) takes in a message string and an error as an Exception type object, and if the error is anything other than "None" it will print the customized notification message "-- Technical Error Message --" and then the error message (from the memory object), the documentation message for the error, the type of the error, with each element separated with a new line using sep='\n'.

This streamlined function will be applied to every section of code with an 'except' in order to handle the error reporting. The first occurrence will be back to the first function created, "read_data_from_file", where the new "output_error_messages" function will be inserted to replace the repetitive lines for error handling.

Function: read_data_from_file (continued)

Under the ‘except FileNotFoundError as e:’, the “output_error_messages” can be called in place of the multiple print statement lines with custom messaging along with exact duplicate lines of code for printing the error/error type/documentation. This is done by first calling the class “IO”, “.” and then calling the function – also providing the expected string message parameter. For the error Exception type, it is passed as “e” following the ‘try’ to load the JSON file contents.

- IO.output_error_messages("Text file must exist before running this script!", e)
- Another ‘except’ to provide a message for non-specific cases
- The ‘finally’ attempts to close the file, in the event it was left open in memory
- The function returns the “student_data” list variable contents



```
    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """
        This function reads the data from the file

            ChangeLog: (Who, When, What)
            Dlarson,11.19.2023,Created function
        :param file_name:
        :param student_data:
        :return: student_data
        """
        # When the program starts, read the file data into a list of dictionary rows (table)
        # Extract the data from JSON file
        try:
            file = open(file_name, "r")
            student_data = json.load(file)
            file.close()
            print("Data has been processed!")
        # Provide structured error handling
        except FileNotFoundError as e:
            IO.output_error_messages("Text file must exist before running this script!", e)
        except Exception as e:
            IO.output_error_messages("There was a non-specific error!", e)
        finally:
            if file.closed == False:
                file.close()
        return student_data
```

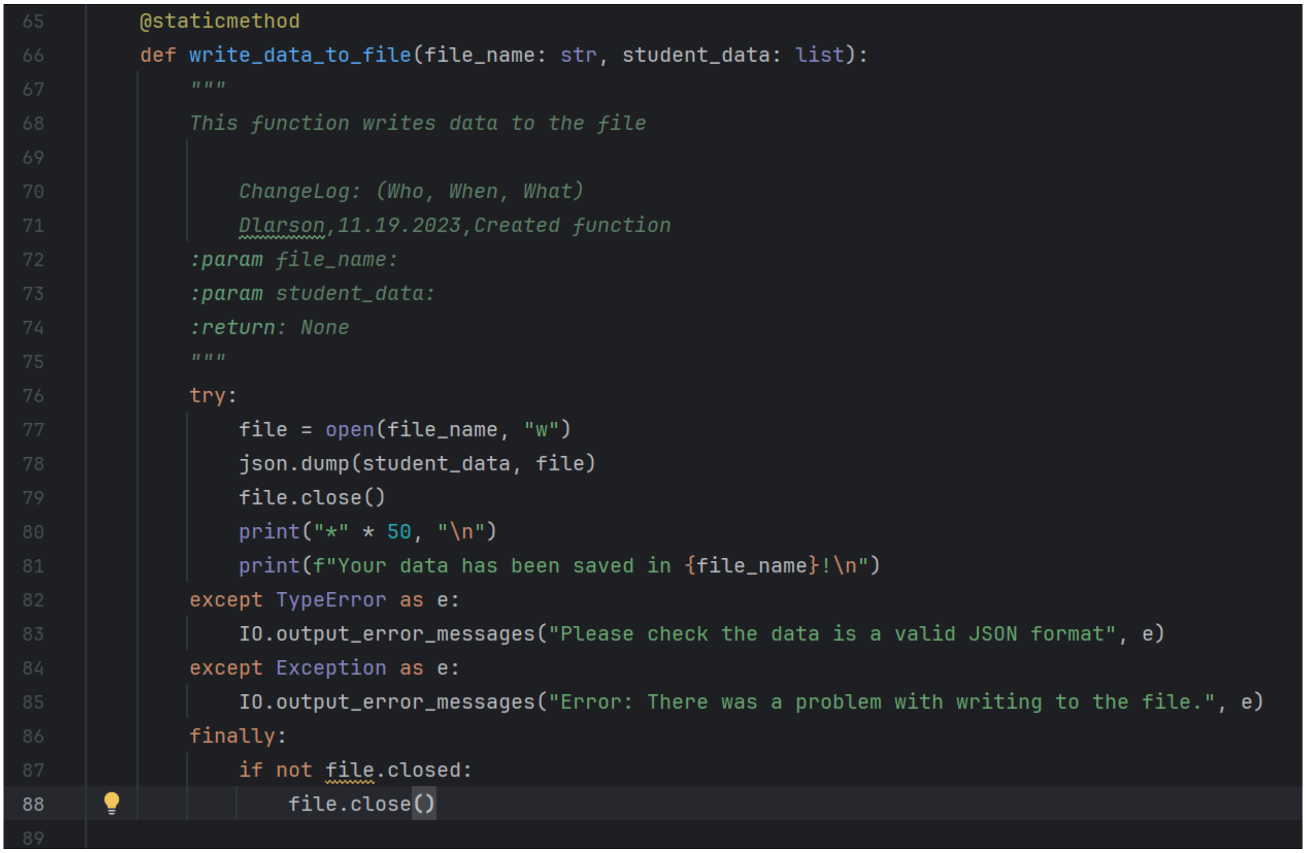
Figure 1.7: The streamlined error reporting is now a single line of code, calling the new Class and Function, within the JSON reader function itself.

Function: write_data_to_file

The last function within the FileProcessor Class is “write_data_to_file”, and is very similar to the “read_data_from_file” method.

- Opens the file argument passed into the function call in “write” mode
- Writes the contents of the student_data list to the file using the json.dump(student_data, file) method

- Customized print statement to display the data was saved
- Error handling is again using the new class/function
  - IO.output_error_messages("…")
- Using the 'finally' to close the file with the "if not" and file.closed the Boolean condition
  - If the file is not already closed, close the file



```python
    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):
        """
        This function writes data to the file

            ChangeLog: (Who, When, What)
            Dlarson,11.19.2023,Created function
        :param file_name:
        :param student_data:
        :return: None
        """
        try:
            file = open(file_name, "w")
            json.dump(student_data, file)
            file.close()
            print("*" * 50, "\n")
            print(f"Your data has been saved in {file_name}!\n")
        except TypeError as e:
            IO.output_error_messages("Please check the data is a valid JSON format", e)
        except Exception as e:
            IO.output_error_messages("Error: There was a problem with writing to the file.", e)
        finally:
            if not file.closed:
                file.close()
```

Figure 1.8: Streamlined code for writing the contents of the list back to the file

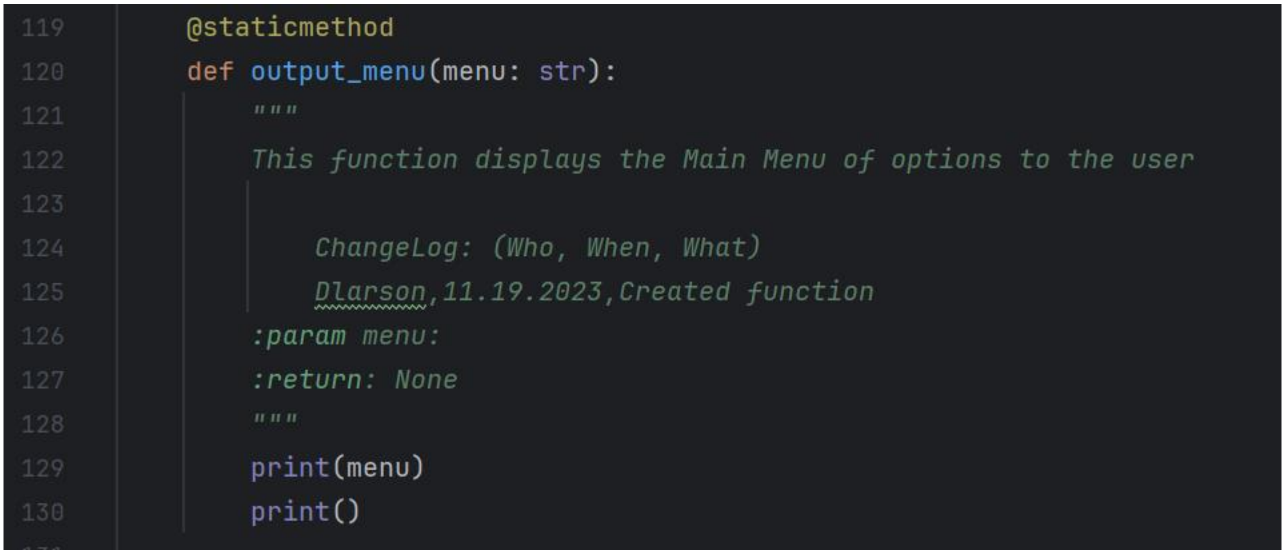
Presenting the Data: Class "IO"

Function: output_error_messages (*see above)

Function: output_menu

The new Class responsible for all Input and Output functions, "IO", is already created with the first function "output_error_messages" being used in the File Processing functions. Next will be the function to display the Main Menu of options to the user: "output_menu" which takes in a string variable as "menu".

- Prints the string "menu" variable passed when the function is called
- Prints an empty line for improved visual formatting



```python
    @staticmethod
    def output_menu(menu: str):
        """
        This function displays the Main Menu of options to the user

            ChangeLog: (Who, When, What)
            Dlarson,11.19.2023,Created function
        :param menu:
        :return: None
        """
        print(menu)
        print()
```

Figure 1.9: This function will display the string variable passed as "menu", along with an added empty line.

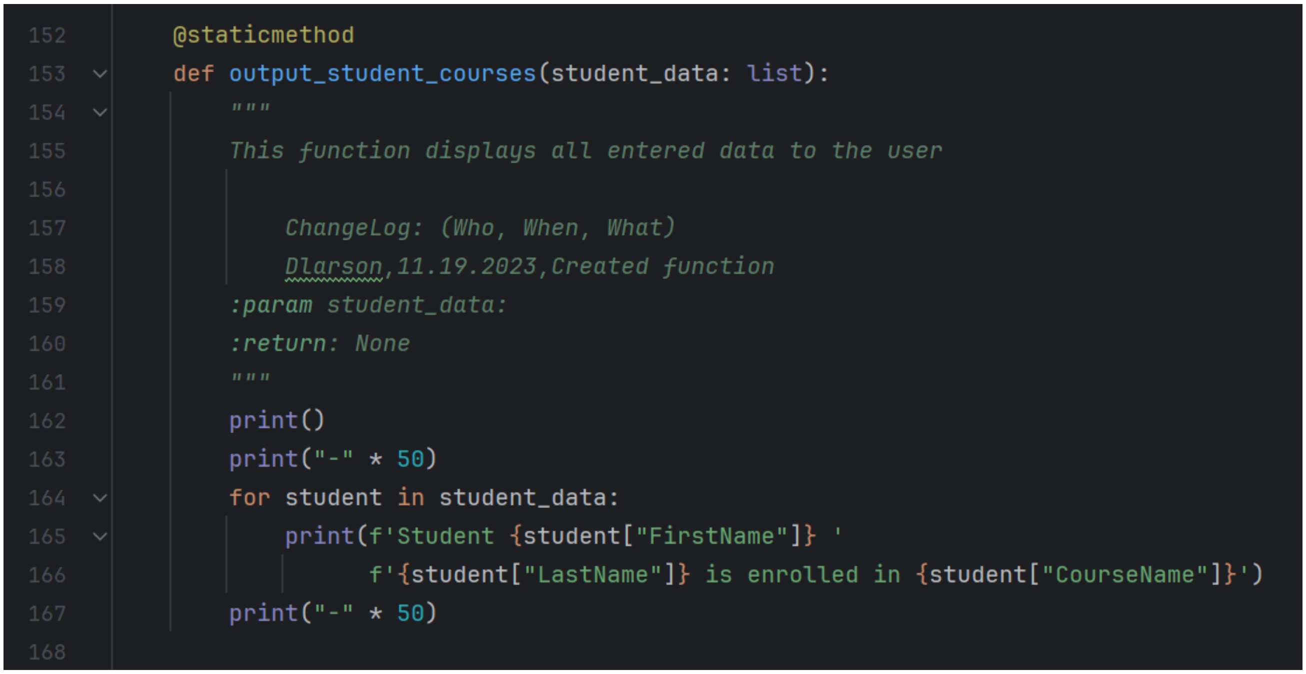
Function: input_menu_choice

- Defines new local variable "choice" and sets to "0"
- Prompts the user for the menu choice and stores it into the 'choice' variable
- Nested within a 'try – except' to include the safeguard for any entries outside of the valid menu options
  - "if" the choice variable is not within the explicitly defined string matches (1,2,3,4), it will raise an exception
  - The exception will print a customized message and then use the IO.output_error_messages to print the error documentation as a string
- Returns the 'choice' variable

Function: output_student_courses

To print all the data, both entered by the user and read from the JSON file, the "student_data" List passed into the function is cycled through using a for-loop. For each entry "student" in student_data, a print statement is formatted (using the f-string method) inserting the dictionary elements into the proper slots using the matching Key value.

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.



```
    @staticmethod
    def output_student_courses(student_data: list):
        """
        This function displays all entered data to the user

            ChangeLog: (Who, When, What)
            Dlarson,11.19.2023,Created function
        :param student_data:
        :return: None
        """
        print()
        print("-" * 50)
        for student in student_data:
            print(f'Student {student["FirstName"]} '
                  f'{student["LastName"]} is enrolled in {student["CourseName"]}')
        print("-" * 50)

```

Figure 1.10: Displaying the data from the file and anything entered by the user.

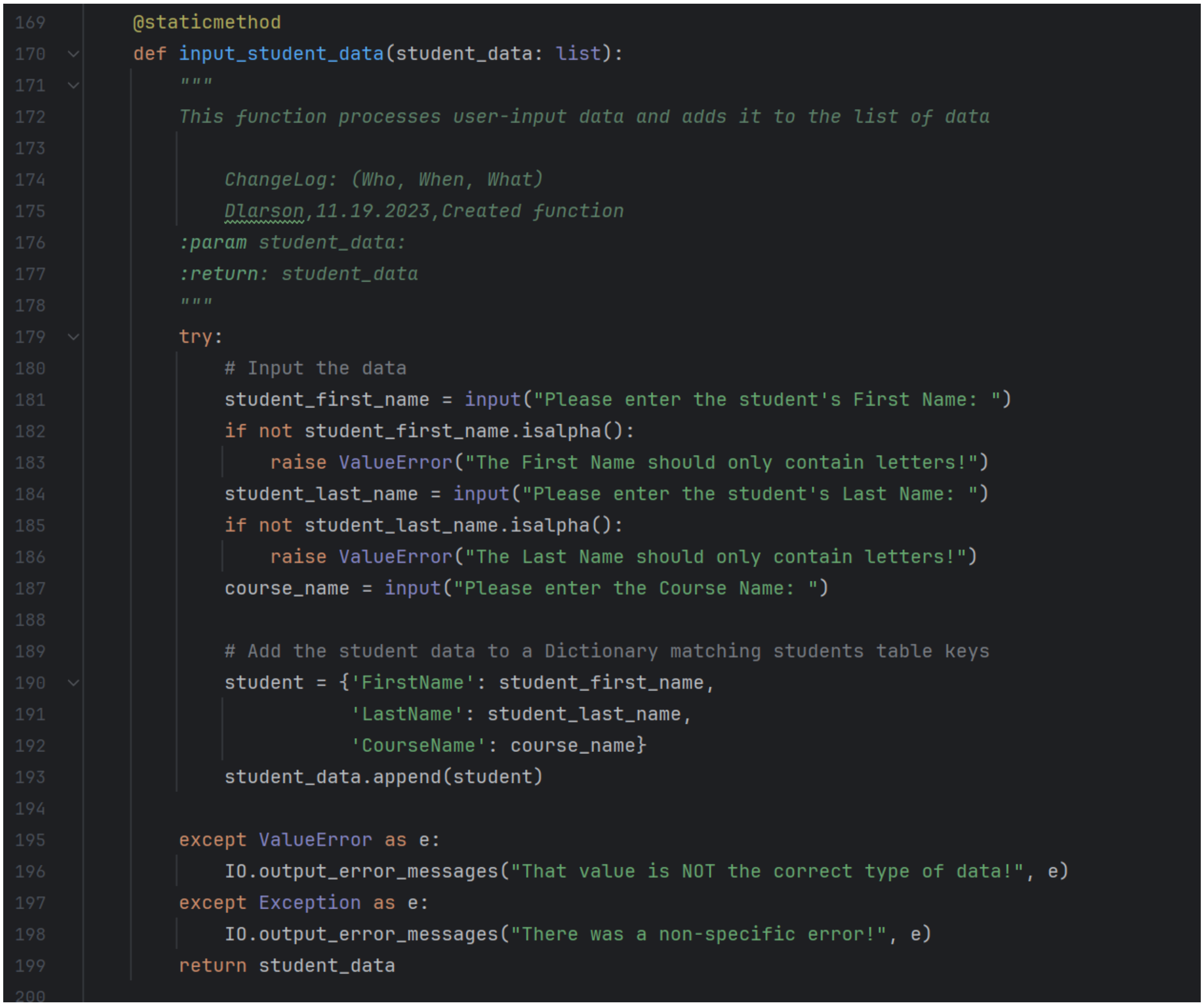
Function: input_student_data

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 a local dictionary named "student", which are appended to the student_data list.

- student = {'FirstName': student_first_name, 'LastName': student_last_name, 'CourseName': course_name}
    - student_data.append(student)
- use IO.output_error_messages to report all error messages



```python
    @staticmethod
    def input_student_data(student_data: list):
        """
        This function processes user-input data and adds it to the list of data

            ChangeLog: (Who, When, What)
            Dlarson,11.19.2023,Created function
        :param student_data:
        :return: student_data
        """
        try:
            # Input the data
            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 = {'FirstName': student_first_name,
                       'LastName': student_last_name,
                       'CourseName': course_name}
            student_data.append(student)

        except ValueError as e:
            IO.output_error_messages("That value is NOT the correct type of data!", e)
        except Exception as e:
            IO.output_error_messages("There was a non-specific error!", e)
        return student_data
```

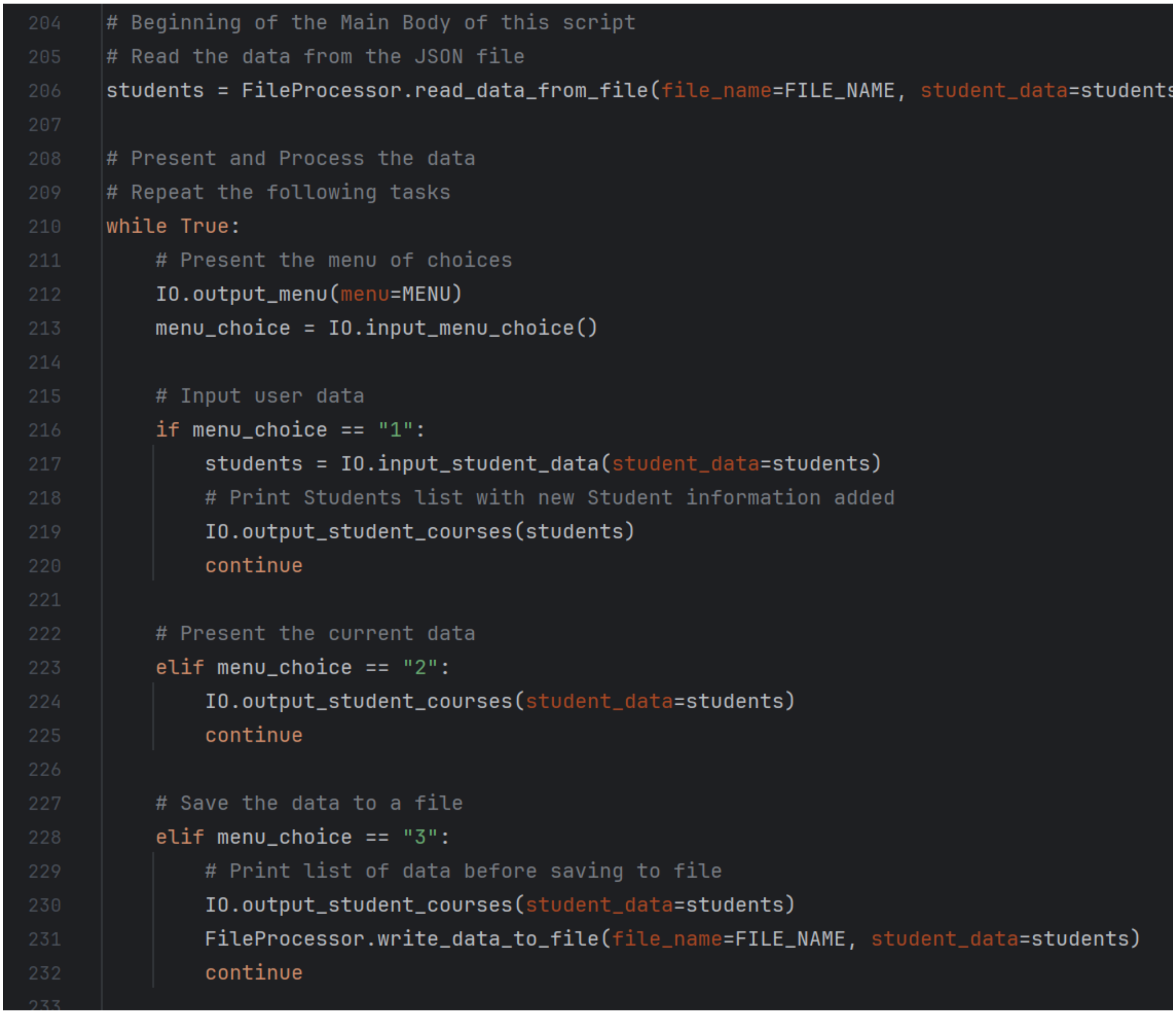
Figure 1.11: User data is captured and added to the "student_data" List, with Customized Error Handling

Main Body

In the body of the script, all classes and functions will be executed. With each if / elif in the 'while' loop, the appropriate function will be called with arguments passed through parameters.

- Call FileProcessor class, read_data_from_file and pass the constant FILE_NAME as file_name and students as the student_data list
    - This will read the file specified as FILE_NAME and put the contents into a list, which is assigned to and returned as student_data
- Create the 'while True:' loop
- Print the menu with IO class output_menu, passing the MENU constant as "menu" argument
- Prompt the user for the menu choice using IO.input_menu_choice() and then assign the result to the variable "menu_choice"
- "if" menu choice == "1":

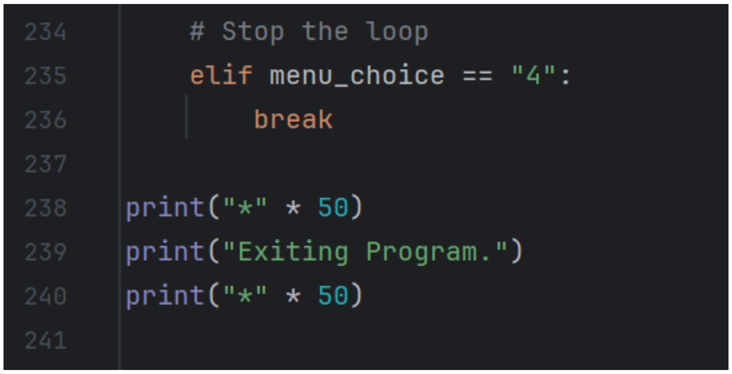
    - Run IO.input_student_data with students passed as student_data
        - This will prompt for user input and append to the student_data list, which is returned and assigned to the ‘students’ variable
    - IO.output_student_courses is added as a user experience enhancement, to automatically print the students list to screen, including the newly-entered student data
- “elif” menu_choice == “2”:
    - Use IO.output_student_courses (passing students list as student_data argument) to print all data – written to file and input by the user – to the screen
- “elif” menu_choice == “3”:
    - Print the list of data before saving using IO.output_student_courses function with students list passed as student_data
    - Write the contents of the students list (passed as student_data) to the file_name specified as the FILE_NAME constant using the FileProcessor class write_data_to_file function



```
# Beginning of the Main Body of this script
# Read the data from the JSON file
students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students

# Present and Process the data
# Repeat the following tasks
while True:
    # Present the menu of choices
    IO.output_menu(menu=MENU)
    menu_choice = IO.input_menu_choice()

    # Input user data
    if menu_choice == "1":
        students = IO.input_student_data(student_data=students)
        # Print Students list with new Student information added
        IO.output_student_courses(students)
        continue

    # Present the current data
    elif menu_choice == "2":
        IO.output_student_courses(student_data=students)
        continue

    # Save the data to a file
    elif menu_choice == "3":
        # Print list of data before saving to file
        IO.output_student_courses(student_data=students)
        FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students)
        continue
```

Figure 1.12: Calling each created Class and Function for the user-selected tasks.

- The loop is stopped in the final “elif” statement to break the “while” condition
- Print statements to inform the user of the program exit



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

print("*" * 50)
print("Exiting Program.")
print("*" * 50)

```

Figure 1.13: Breaking the loop and displaying a message to the user.

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 “Assignment06.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” was improved through the implementation of Classes and Functions to organize and modularize the code. While most of the functional script remained the same, the ways in which it was reordered / restructured vastly improved the stability, maintainability, and scalability through the separation of elements.

- Classes and Functions to create modular blocks of code to be reused

- Passing and returning variables as arguments for local variables helps control the “side effects” on global scale variables
- Organizing code by using Separations of Concerns