Sarah West

11/21/24

IT FDN 110 A

Assignment 06

https://github.com/sarahmwest/IntroToProg-Python-Mod06

Functions and Classes

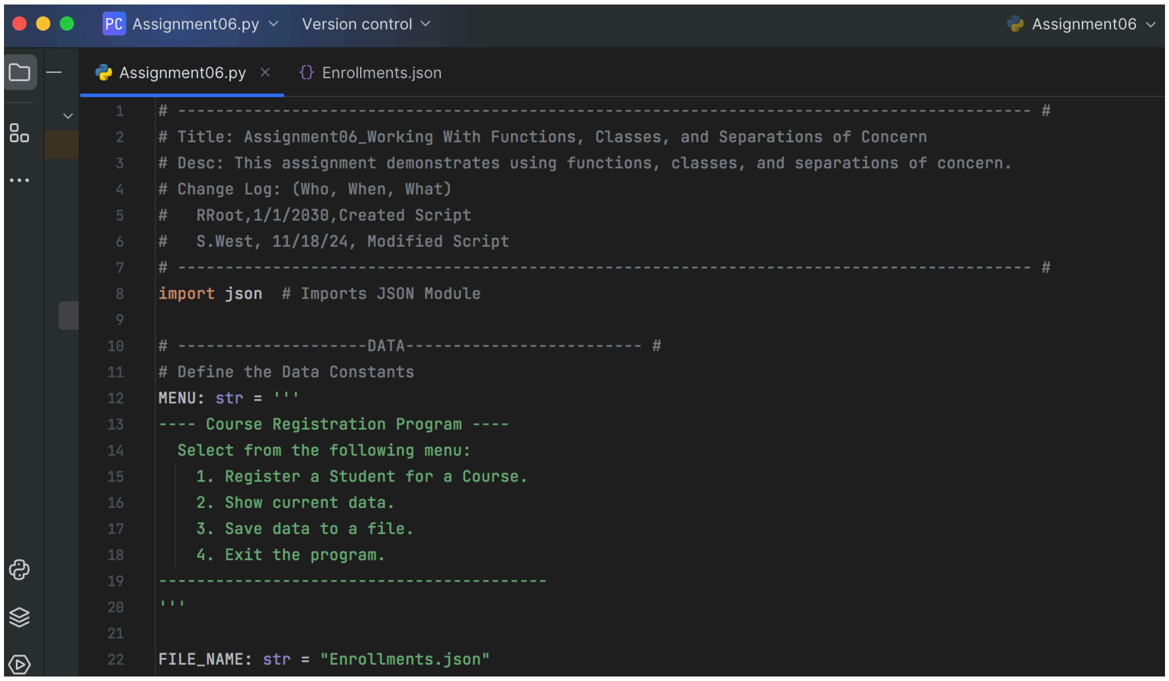
Introduction

In this assignment, I built an interactive program that utilizes a collection of dictionaries and reads and writes data to JSON (JavaScript Object Notation) files similar to the Module 05 assignment. However, in this assignment the code is organized into functions (methods) and classes to improve the readability, maintainability, and usability of the code. Parameters are used to define local variables and arguments are passed into the methods to define the parameters declared in the function. This script features additional organization using the Separation of Concerns pattern to further enhance the maintainability, scalability, and readability of the code.

Creating the Script

Script Header and the Data Layer - Defining the Constants

I began by opening the "Assignment06-starter.py" that was provided us. Some basic pseudo-code and starter code was already written. I updated the script header to contain my information and the constants (**Figure 1**).



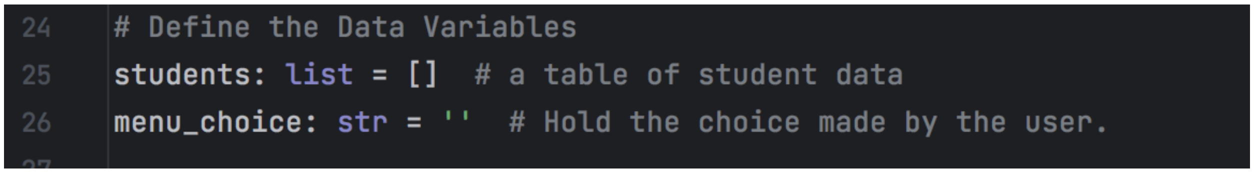
```
# ------------------------------------------------------------------------------------------ #
# Title: Assignment06_Working With Functions, Classes, and Separations of Concern
# Desc: This assignment demonstrates using functions, classes, and separations of concern.
# Change Log: (Who, When, What)
#   RRoot,1/1/2030,Created Script
#   S.West, 11/18/24, Modified Script
# ------------------------------------------------------------------------------------------ #
import json  # Imports JSON Module

# -------------------DATA----------------------- #
# Define the Data Constants
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.
-----------------------------------------
'''

FILE_NAME: str = "Enrollments.json"
```

Figure 1. A screenshot showing the updated script header, constants, and start of the data layer.

In this script, I used the Separation of Concerns (SoC) pattern to organize the code to improve readability, maintainability, and scalability of the code. The SoC pattern organizes code by breaking it down into distinct, self-contained blocks that have different functionalities (i.e., by their "concerns"). Three types of concerns that are commonly used are: 1) presentation concern, which groups code that is associated with how data is displayed (user interfaces, styling, etc.), 2) logic concern, which groups code that is associated with the core functionality of the script, such as data processing, and 3) data storage concern, which deals with code associated with how data is stored and managed. With these in mind, I organized the code such that all the variables and constants declared after the script header are in the "Data Layer", i.e., the #--DATA--# section (**Figure 1**).

Data Layer - Defining the Variables

Defining the variables was simple since the starter file gave us nearly all of them. As you will see later in the script, many were moved to the functions to act as parameters, and many were deleted. As shown in **Figure 2**, the only variables that I kept were students: list=[] and menu_choice: str= ‘’. Declaring variables at the start of the script outside the context of a function means that these are acting as global variables and their scope is the entire script and will be used thoughout. Parameters act as local variables and are used in their assigned functions only. The variables declaration is part of the SoC Data Layer.



```
# Define the Data Variables
students: list = []  # a table of student data
menu_choice: str = ''  # Hold the choice made by the user.
```

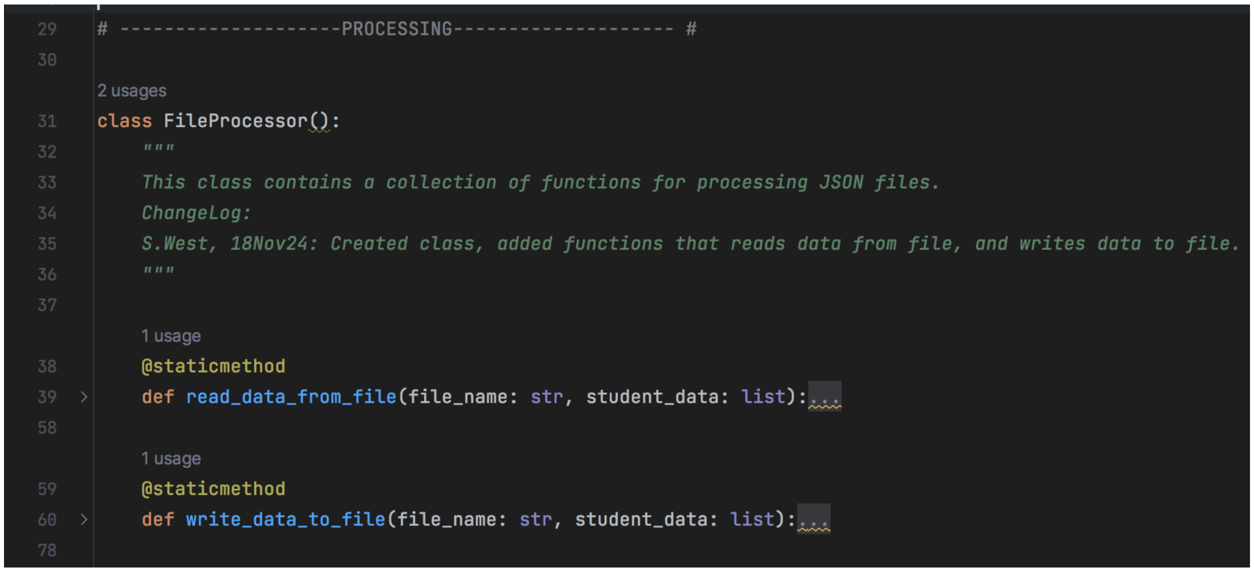
Figure 2. A screenshot showing the updated variables as part of the data layer.

The Processing Layer

FileProcessor Class

Figure 3 shows the SoC Processing Layer, or the "application layer" where the data processing and file management happens. Under the processing layer, I created a class called "FileProcessor" which group together methods that read and write data to and from my JSON file. Classes make it easier to manage code by grouping them together based on functionality or purpose.

Under the class definition, I have a document string, or "docstring" that states the purpose of the class and the change log. Docstrings make it easier to access non-functional code that describes the purpose of the code, such as what parameters a method contains. Hovering your mouse over the code associated with the docstring, like the FileProcessor class in this case, can access the docstring via a popup window, which eliminates the need to locate it in the script, which can be cumbersome for very long scripts. Note that every class and function in my script has an associated docstring.

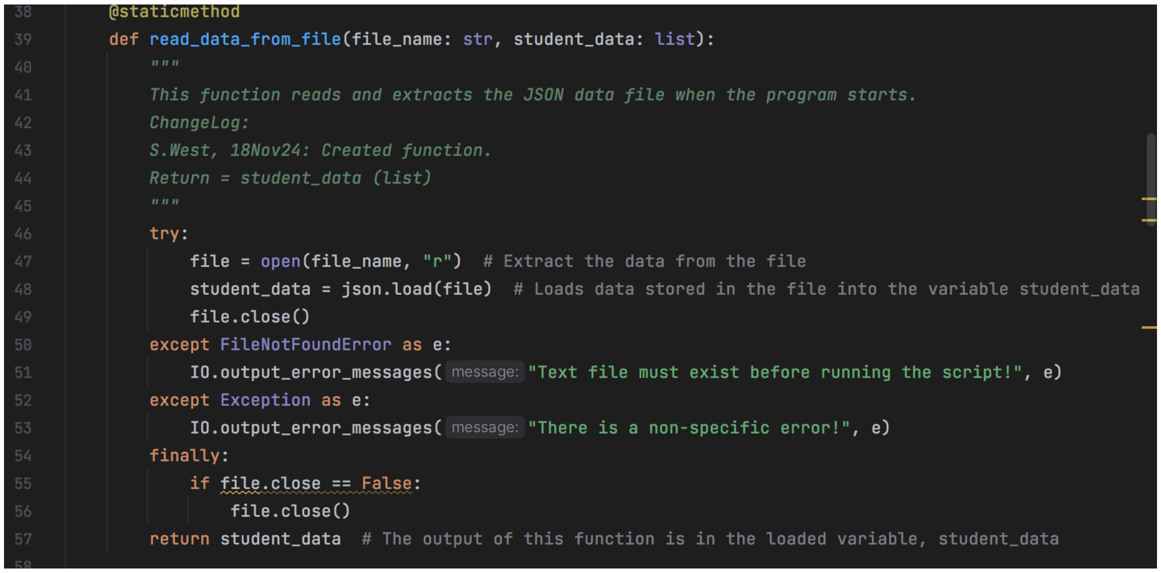


```
# --------------------PROCESSING-------------------- #

2 usages
class FileProcessor():
    """
    This class contains a collection of functions for processing JSON files.
    ChangeLog:
    S.West, 18Nov24: Created class, added functions that reads data from file, and writes data to file.
    """

    1 usage
    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):...

    1 usage
    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):...
```

Figure 3. A screenshot showing the FileProcessor class, the docstring, and the two methods contained within the class.

Method: read_data_from_file

As shown in **Figure 3**, the FileProcessor class contains two custom methods: read_data_from_file and write_data_from_file. The first method, read_data_from_file, which is collapsed in **Figure 3**, is used to read the data from the JSON file and load it into the parameter 'student_data'. The "@staticmethod" statement is a decorator that tells Python to use this class directly and that the function code doesn't change, or is "static". All my custom functions in this script have the staticmethod decorator. To define a method, the statement "def" is used, followed by the function name. The statements inside the parentheses are the method's parameters, or local variables, that are used within the scope of the method. Using variables as parameters is considered best practise compaired to declaring global and local variables because it provides encapsulation, which is less error prone, provides more flexibility for the function, and gives it predictable behavior. **Figure 4** shows the expanded function and its code.



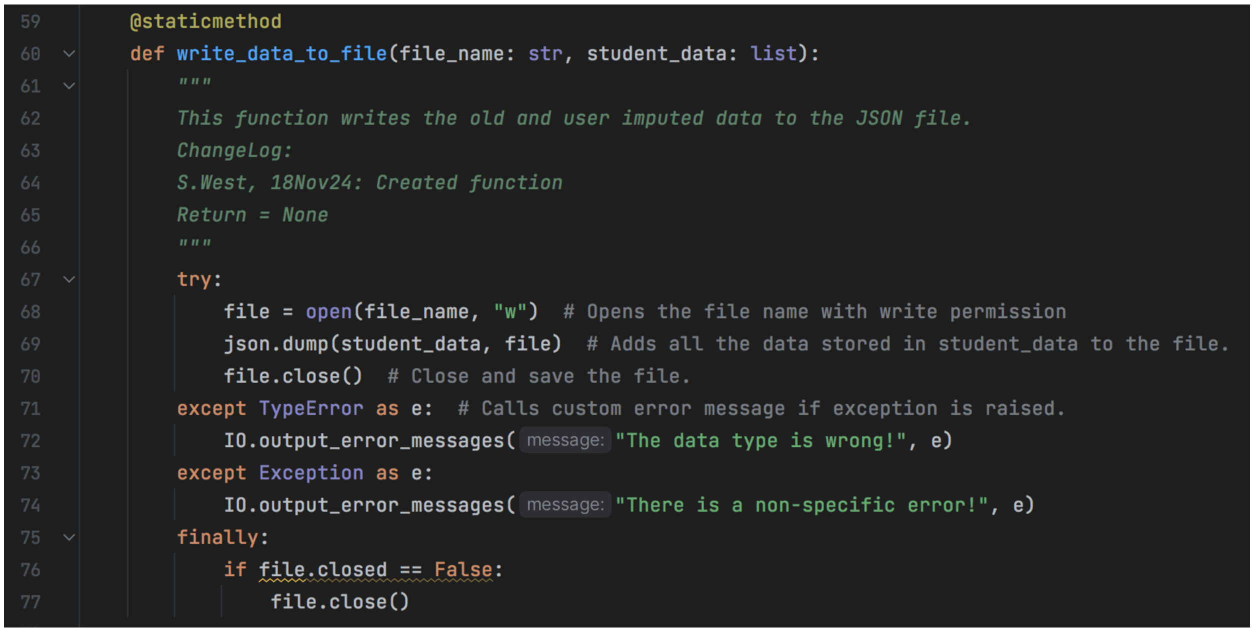
```
    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """
        This function reads and extracts the JSON data file when the program starts.
        ChangeLog:
        S.West, 18Nov24: Created function.
        Return = student_data (list)
        """
        try:
            file = open(file_name, "r")  # Extract the data from the file
            student_data = json.load(file)  # Loads data stored in the file into the variable student_data
            file.close()
        except FileNotFoundError as e:
            IO.output_error_messages( message: "Text file must exist before running the script!", e)
        except Exception as e:
            IO.output_error_messages( message: "There is a non-specific error!", e)
        finally:
            if file.close == False:
                file.close()
        return student_data  # The output of this function is in the loaded variable, student_data
```

Figure 4. A screenshot showing the read_data_from_file method and its docstring.

In the read_data_from_file method shown in **Figure 4**, I declared the parameters file_name and student_data without arguments. I have nested the code into a try/except block to handle potential errors reading the file, such as a FileNotFound error and a generic Exception. The messages that are displayed to the user when these errors are called are passed into the method along with the the object "e" which gives basic information about the error. I will discuss this idea more in a later section. Note that the except blocks call a separate class.method that I defined later on and will discuss them in detail in a later section. The try block contains basic code similar to Assignment 05 that opens a JSON file, reads the file, and stores the data from the file in the parameter student_data. The significance of this method is the 'return' value; the output of calling this method is the variable student_data which stores the data read from the JSON file. For simplicity, I will continue to discuss each class and method sequentially in this knowledge document and discuss the body of the script last where the functions are called and the return values are used.

Method: write_data_to_file

Figure 5 shows the method I created to write the data to a file, which is called after new user input is collected. In this method, I delcared the same parameters I used in the read_data_from_file method discussed above. I used similar code to what was used in Assignment 5 to open the JSON file, write all the information stored in the student_data variable/parameter, and then to close the file and save the data. I nested this code into try/except blocks to handle any errors, such as a TypeError or generic Exception. Note that this method does not have a return value, which means the output of the function does not return any information in a variable.

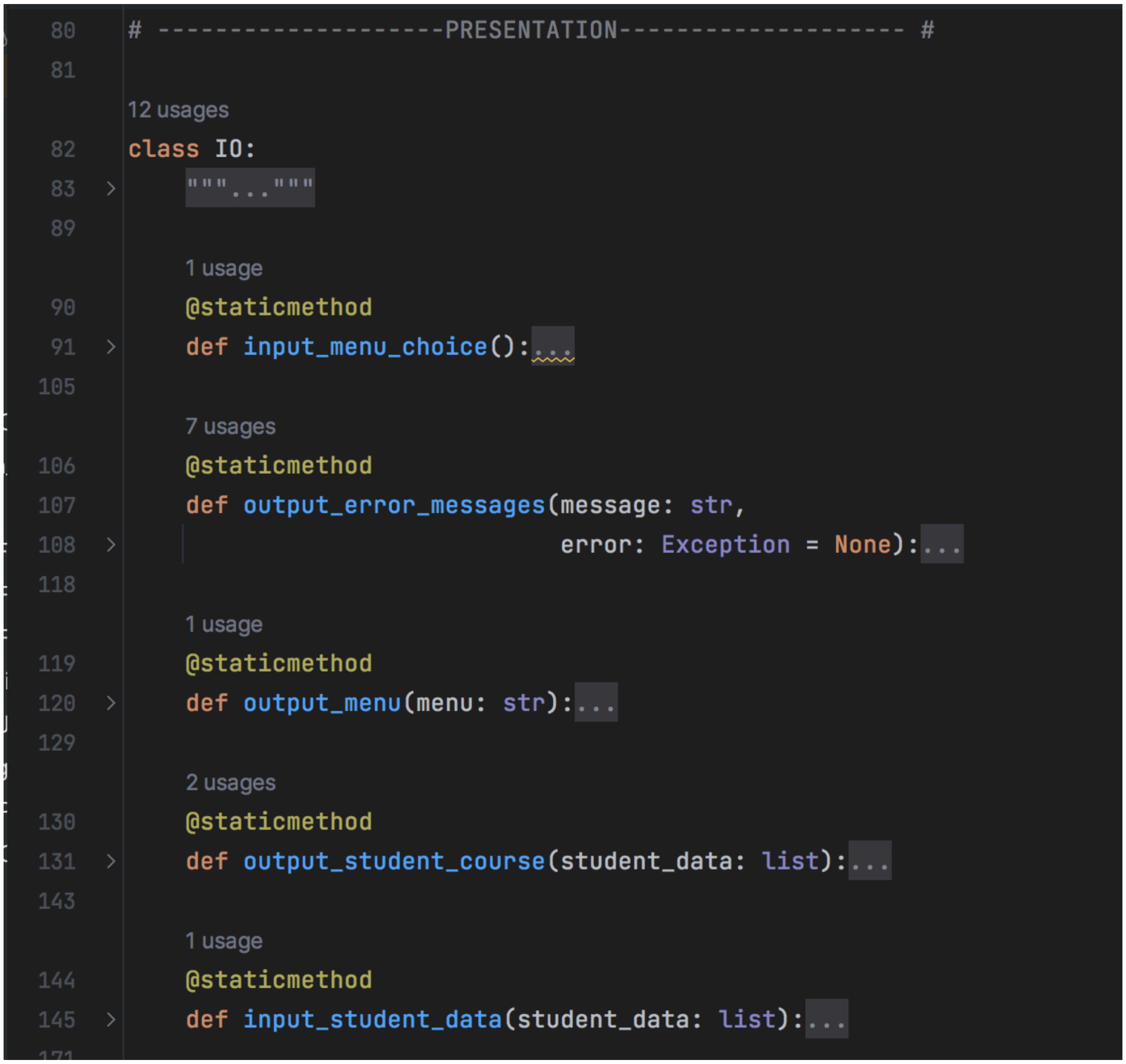


```
    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):
        """
        This function writes the old and user imputed data to the JSON file.
        ChangeLog:
        S.West, 18Nov24: Created function
        Return = None
        """
        try:
            file = open(file_name, "w")  # Opens the file name with write permission
            json.dump(student_data, file)  # Adds all the data stored in student_data to the file.
            file.close()  # Close and save the file.
        except TypeError as e:  # Calls custom error message if exception is raised.
            IO.output_error_messages( message: "The data type is wrong!", e)
        except Exception as e:
            IO.output_error_messages( message: "There is a non-specific error!", e)
        finally:
            if file.closed == False:
                file.close()
```

Figure 5. A screenshot showing the definition of the write_data_to_file method.

The Presentation Layer

Figure 6 shows the beginning of the SoC Presentation Layer which groups the code that controls the input and output of user data. Note that the docstring associated with the IO class is collaped to show all the methods imbedded in this class.



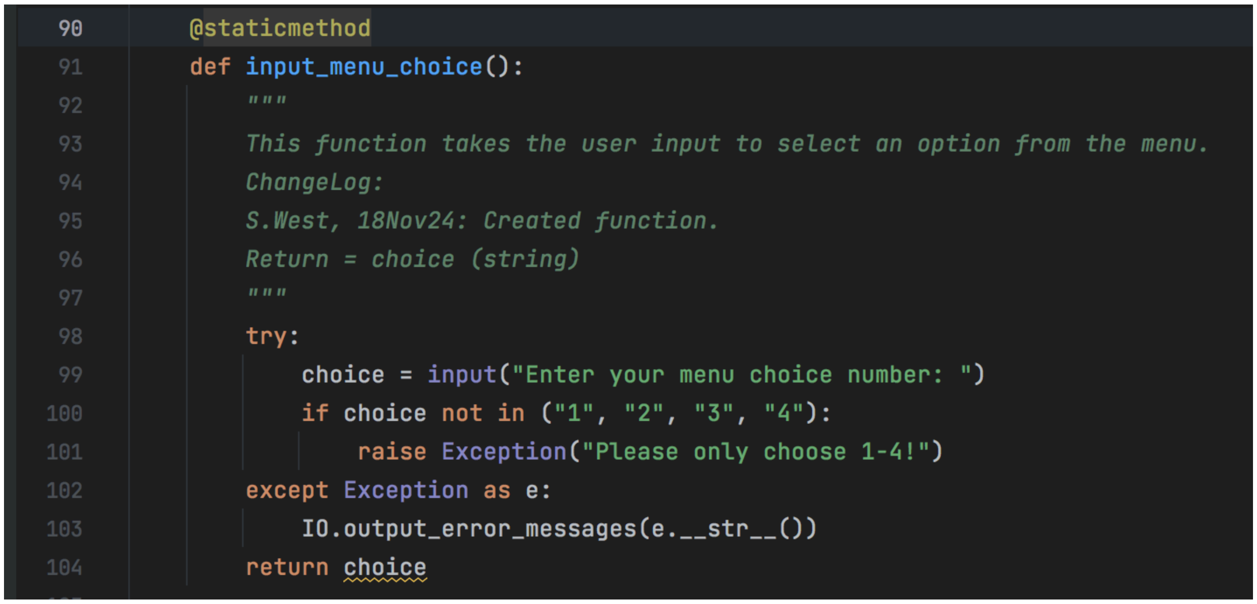
```
# -------------------PRESENTATION------------------- #

12 usages
class IO:
    """..."""

    1 usage
    @staticmethod
    def input_menu_choice():...

    7 usages
    @staticmethod
    def output_error_messages(message: str,
                              error: Exception = None):...

    1 usage
    @staticmethod
    def output_menu(menu: str):...

    2 usages
    @staticmethod
    def output_student_course(student_data: list):...

    1 usage
    @staticmethod
    def input_student_data(student_data: list):...
```

Figure 6. A screenshot showing the IO class and the methods contained in the class.

Method: input_menu_choice

The input_menu_choice method in **Figure 7** is imbedded in a try/except block and captures the user input when they select an option from the menu. There are no parameters needed for this function, and the function returns the variable "choice" which contains a string of either "1", "2", "3", or "4", depending on the user input. I raised an Exception if the user input is not 1-4, and the

Exception will display the error captured as the object “e”, which is discussed below, and the docstring associated with the error.

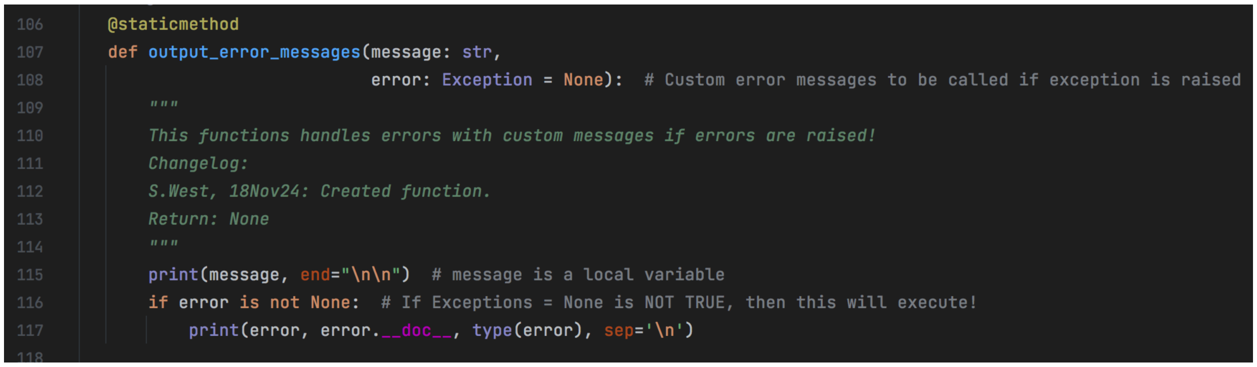


```python
    @staticmethod
    def input_menu_choice():
        """
        This function takes the user input to select an option from the menu.
        ChangeLog:
        S.West, 18Nov24: Created function.
        Return = choice (string)
        """
        try:
            choice = input("Enter your menu choice number: ")
            if choice not in ("1", "2", "3", "4"):
                raise Exception("Please only choose 1-4!")
        except Exception as e:
            IO.output_error_messages(e.__str__())
        return choice
```

Figure 7. A screenshot showing the input_menu_choice method.

Method: output_error_messages

The error handling is done through a custom method called output_error_messages shown in **Figure 8.** Here, I declared the parameter ‘message’ of type: string that takes on different values depending on the method that calls the output_error_messages method, such as in **Figure 5**. Additionally, I defined the parameter ‘error’, which is an Exception, and passed it the argument of ‘None’. This means that the method automatically assumes there are no exceptions unless I pass an argument to an exception. For example, in **Figure 5** above in the TypeError except block, I first passed the parameter ‘message’ the argument: “The data type is wrong!”, followed by the object ‘e’. The first argument passed has to be the argument for the ‘message’ parameter I declared in the output_error_messages method, and the second argument passed in the TypeError exception has to be the error of type: Exception. If only one argument is passed when the output_error_messages method is called, Python will assume it is the first parameter ‘message’, and error: Exception = None. However, if I pass a second argument when I call the output_error_messages method (i.e., the object e), then the ‘if’ block will execute because then error: Exception no longer equals ‘None’, but rather equals ‘e’.

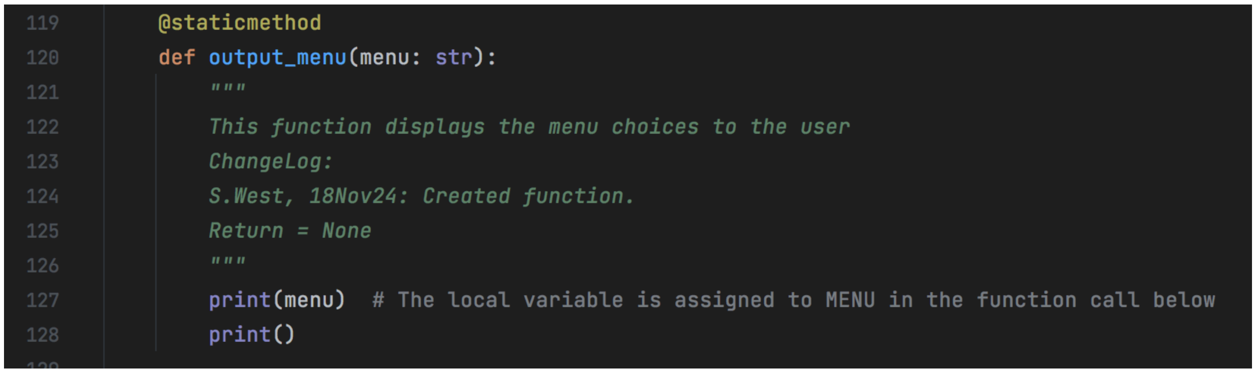


```python
    @staticmethod
    def output_error_messages(message: str,
                              error: Exception = None):  # Custom error messages to be called if exception is raised
        """
        This functions handles errors with custom messages if errors are raised!
        Changelog:
        S.West, 18Nov24: Created function.
        Return: None
        """
        print(message, end="\n\n")  # message is a local variable
        if error is not None:  # If Exceptions = None is NOT TRUE, then this will execute!
            print(error, error.__doc__, type(error), sep='\n')
```

Figure 8. A screenshot of the block of code for the error handling method.

Method: output_menu

In Figure 9, the output_menu method is defined, which is rather simple. Here, I am declaring the parameter "menu" of type string and simply printing the menu for the user.

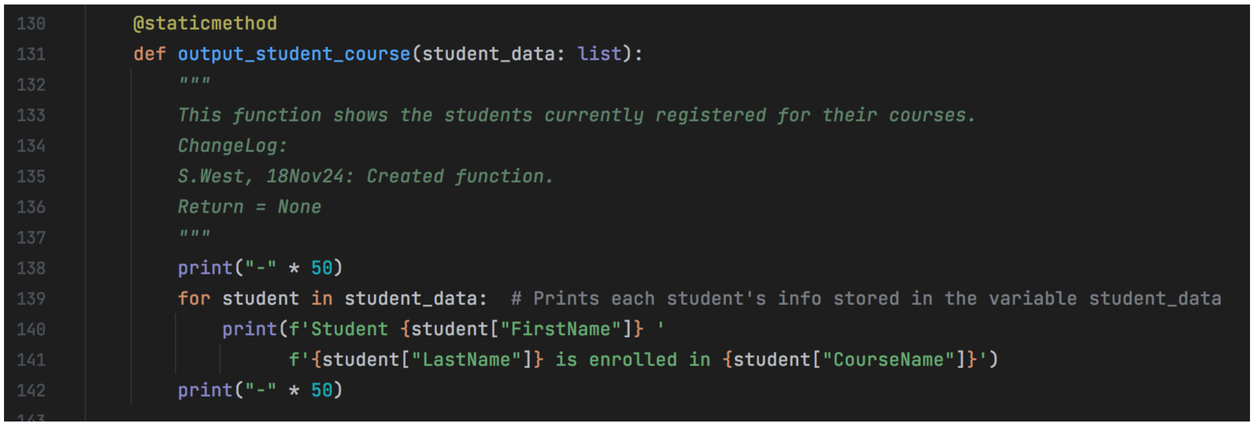


```
    @staticmethod
    def output_menu(menu: str):
        """
        This function displays the menu choices to the user
        ChangeLog:
        S.West, 18Nov24: Created function.
        Return = None
        """
        print(menu)  # The local variable is assigned to MENU in the function call below
        print()
```

Figure 9. A screenshot of the block of code for the output_menu function.

Method: output_student_course

Figure 10 shows the method for printing out the data stored in the parameter student_data. The code uses a for loop to print out each student's data that is stored in the parameter student_data. It is important to note that all the data stored in the student_data parameter is returned in the method input_student_data discussed below, as well as the data returned from the above function read_data_from_file. I will discuss this further below when I discuss the function calls.

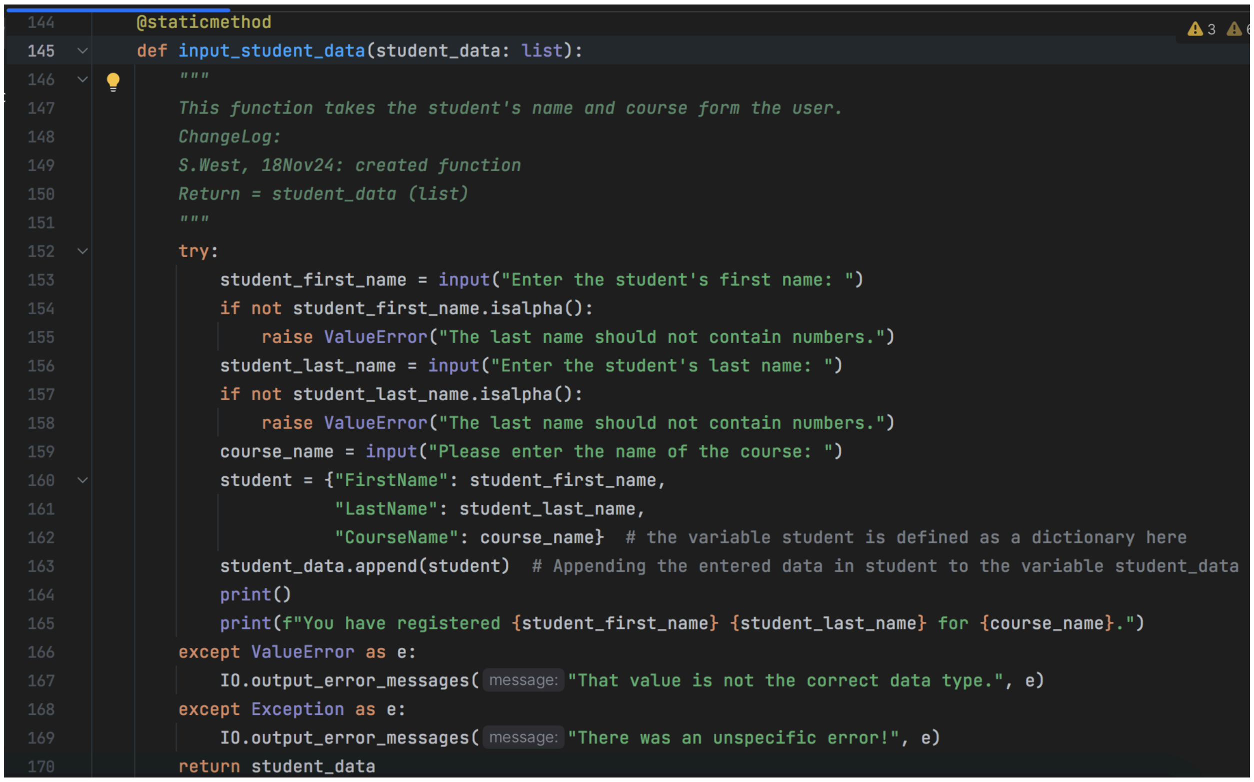


```
    @staticmethod
    def output_student_course(student_data: list):
        """
        This function shows the students currently registered for their courses.
        ChangeLog:
        S.West, 18Nov24: Created function.
        Return = None
        """
        print("-" * 50)
        for student in student_data:  # Prints each student's info stored in the variable student_data
            print(f'Student {student["FirstName"]} '
                  f'{student["LastName"]} is enrolled in {student["CourseName"]}')
        print("-" * 50)
```

Figure 10. A screenshot of the block of code for the output_student_course method.

Method: input_student_data

The last custom method in the IO class is the input_student_data method shown in **Figure 11** that captures the user input and stores it in the parameter student_data. I also added a print statement to let the user know the data was successfully entered. I nested the code in a try/except block to catch some errors such as entering the wrong type of data (not alphanumeric inputs for first and last name) as well as a generic Exception. After the first and last name and course name are inputted, the data is captured in the variable 'student' and formatted into a dictionary list (suitable for JSON files) and appended to the parameter ‘student_data’, which means that the parameter now has the newly inputted data as well as the original data in the JSON file that was read at the initiation of the program. This method returns the variable student_data now loaded with the newly inputted data.



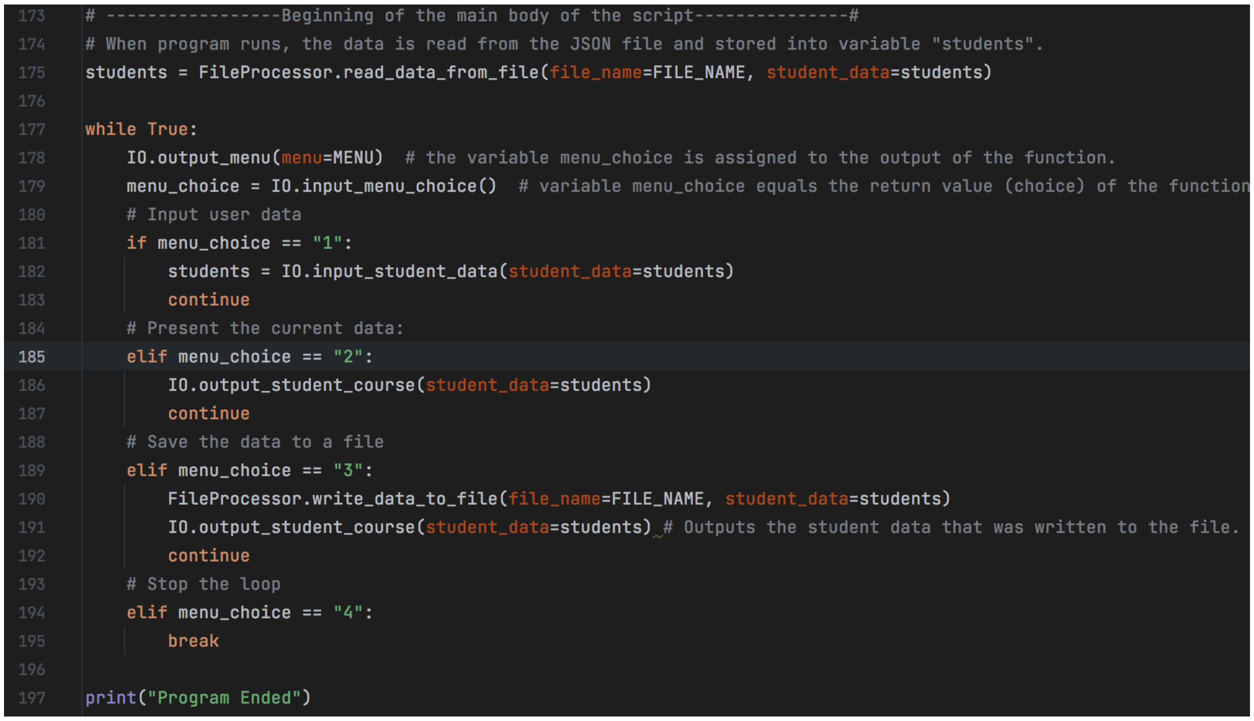
```python
    @staticmethod
    def input_student_data(student_data: list):
        """
        This function takes the student's name and course form the user.
        ChangeLog:
        S.West, 18Nov24: created function
        Return = student_data (list)
        """
        try:
            student_first_name = input("Enter the student's first name: ")
            if not student_first_name.isalpha():
                raise ValueError("The last name should not contain numbers.")
            student_last_name = input("Enter the student's last name: ")
            if not student_last_name.isalpha():
                raise ValueError("The last name should not contain numbers.")
            course_name = input("Please enter the name of the course: ")
            student = {"FirstName": student_first_name,
                       "LastName": student_last_name,
                       "CourseName": course_name}  # the variable student is defined as a dictionary here
            student_data.append(student)  # Appending the entered data in student to the variable student_data
            print()
            print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
        except ValueError as e:
            IO.output_error_messages(message: "That value is not the correct data type.", e)
        except Exception as e:
            IO.output_error_messages(message: "There was an unspecific error!", e)
        return student_data
```

Figure 11. A screenshot of the input_student_data method.

Main Body of the Script

Method Calls

After all the method definitions and variable/constant declarations that were in the SoC data layer, processing layer, and presentation layer, the main body of the script begins. Here, I call the classes and their methods by using the **class.method(parameter = argument)** statements shown in **Figure 12**. When the program is initiated, I want the JSON file to be opened, read, and the data to be stored in the global variable ‘students’. To do this, I call the read_data_from_file method in the FileProcessor class and give arguments to the parameters I previously declared in the method. For example, the parameter file_name is passed the argument FILE_NAME that points the parameter to the constant that I declared in the data layer. This tells the function to use that file when running. Since this method returns the parameter student_data, I have to pass the argument ‘students’ to the parameter. Furthermore, since I want the output of the function to be stored in the global variable ‘students’ for use in other methods, I set the output of the method call to equal the variable ‘students’. This passes the information stored in the parameter student_data to the variable ‘students’ so it’s in scope of the entire script.



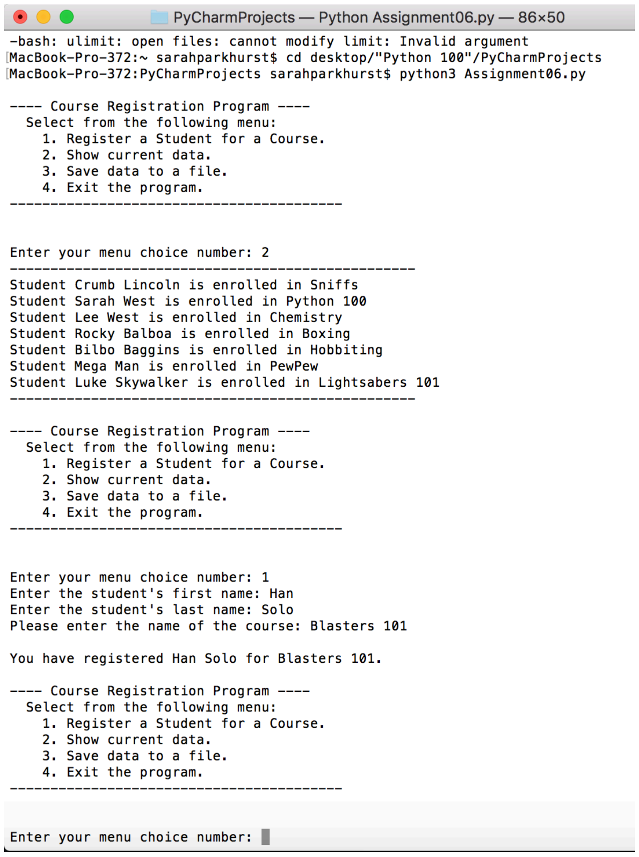
```
# -----------------Beginning of the main body of the script--------------#
# When program runs, the data is read from the JSON file and stored into variable "students".
students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)

while True:
    IO.output_menu(menu=MENU)  # the variable menu_choice is assigned to the output of the function.
    menu_choice = IO.input_menu_choice()  # variable menu_choice equals the return value (choice) of the function
    # Input user data
    if menu_choice == "1":
        students = IO.input_student_data(student_data=students)
        continue
    # Present the current data:
    elif menu_choice == "2":
        IO.output_student_course(student_data=students)
        continue
    # Save the data to a file
    elif menu_choice == "3":
        FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students)
        IO.output_student_course(student_data=students) # Outputs the student data that was written to the file.
        continue
    # Stop the loop
    elif menu_choice == "4":
        break

print("Program Ended")
```

Figure 12. A screenshot showing the main body of the script.

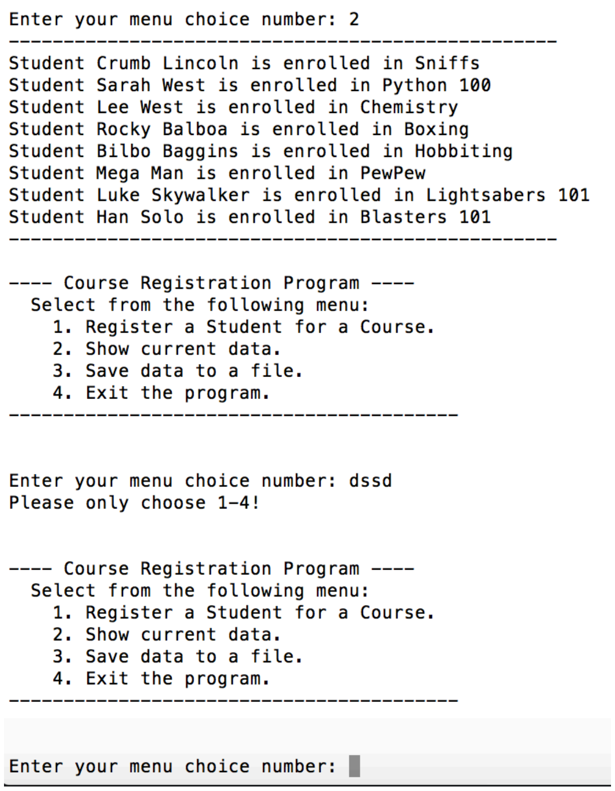
I embedded the rest of the script in a while loop so the program continues to iterate through the options until the user quits the program. I called the methods within each class to execute when the appropriate choice was selected from the menu. For the output_menu call, I assigned the argument MENU to the parameter ‘menu’. I assigned the output of the input_menu_choice method to equal menu_choice since that method returns the parameter ‘choice’, which stores the user’s input. Next, the different options selected by the user are assigned to the various methods using an if-elif-else loop. For menu choice = 1, the input_student_data method runs, in which I assigned the parameter student_data to the argument of ‘students’ along with the output of the method call, so the output of that function stores all inputted data in the variable ‘students’ along with data that was already in the JSON file when the program was initiated. For menu choice 3, I added the data output method along with the write_data_to_file method to show the user that the data was successfully written to the file. Of course, menu choice of 4 ends the program.

Testing the Script in PyCharm and MacOS

I tested the script by running it in PyCharm, and it works great! I also tested the error handling by purposefully entering incorrect names, or changing the file name. After I confirmed the script runs as expected in PyCharm, and I confirmed that the script runs successfully from the terminal on my MacOS, which is shown in **Figure 13** below. I also confirmed that the program correctly writes the data into the JSON file, which it does!



```
-bash: ulimit: open files: cannot modify limit: Invalid argument
MacBook-Pro-372:~ sarahparkhurst$ cd desktop/"Python 100"/PyCharmProjects
MacBook-Pro-372:PyCharmProjects sarahparkhurst$ python3 Assignment06.py

---- 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.
-----------------------------------------


Enter your menu choice number: 2
--------------------------------------------------
Student Crumb Lincoln is enrolled in Sniffs
Student Sarah West is enrolled in Python 100
Student Lee West is enrolled in Chemistry
Student Rocky Balboa is enrolled in Boxing
Student Bilbo Baggins is enrolled in Hobbiting
Student Mega Man is enrolled in PewPew
Student Luke Skywalker is enrolled in Lightsabers 101
--------------------------------------------------

---- 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.
-----------------------------------------


Enter your menu choice number: 1
Enter the student's first name: Han
Enter the student's last name: Solo
Please enter the name of the course: Blasters 101

You have registered Han Solo for Blasters 101.

---- 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.
-----------------------------------------


Enter your menu choice number:
```



```
Enter your menu choice number: 2
--------------------------------------------------
Student Crumb Lincoln is enrolled in Sniffs
Student Sarah West is enrolled in Python 100
Student Lee West is enrolled in Chemistry
Student Rocky Balboa is enrolled in Boxing
Student Bilbo Baggins is enrolled in Hobbiting
Student Mega Man is enrolled in PewPew
Student Luke Skywalker is enrolled in Lightsabers 101
Student Han Solo is enrolled in Blasters 101
--------------------------------------------------

---- 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.
-----------------------------------------


Enter your menu choice number: dssd
Please only choose 1-4!


---- 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.
-----------------------------------------


Enter your menu choice number:
```

Figure 13. Screenshots showing the output of the script running in MacOS terminal.

Summary

In this assignment, I successfully created a python script that takes a starting JSON file, collects new information inputted by the user, and adds that information to the JSON file without deleting the old information. I used the script that was provided to us with preliminary code and pseudo-code and added my code to that script. I organized my script according to the best practice of separation of concerns, and embedded the code into classes and custom methods to maximize the organization, readability, and usability. I am happy to report that the script runs as intended!