Geoff DuBuque

August 6, 2024

IT FDN 110 A Su 24

Assignment 06

https://github.com/gdubuque/IntroToProg-Python-Mod06.git

Assignment 06 – Functions

Introduction

This module's programming assignment keeps all the same functionality as the previous program, to collect user data and display messages about registering a student for a class, but introduces 3 new techniques for reorganizing the script, aka **refactoring**. These techniques included using functions, classes, and the separation of concerns (SoC) pattern. Refactoring the script in this way adds many benefits such as modularity, reusability, encapsulation, flexibility, maintainability, ease of collaboration, and many others. The following sections describe the steps I used to complete the programming task of this assignment. All these programming ideas and syntax can be found in the module's notes, labs, and starter Python file, referenced at the end of this paper.

Creating the Python script

Script Header and Program Data Layer

As usual we start with the assignment's starter Python script that was a complete functioning program form the previous assignment. The script starts with our standard header, updated with my changes in the change log, and any imports we need to make. The first step involves reorganizing the script into different areas of concern, or layers, regarding the program's functionality. The first layer, the "Data Layer", gathers the program's data, including the global constants and variables, meant to be used for storing and processing data. These are variables intended to be used throughout the program in any layer. All other variables were removed because they only need to be used locally to a function. See **Figure 1** on the next page for the data layer of the script.

Assignment06.py

```python
# ------------------------------------------------------------------------------------------ #
# Title: Assignment06
# Desc: This assignment demonstrates using functions with structured error handling,
#       adding the use of methods, classes and Separation of Concerns pattern
# Change Log: (Who, When, What)
#   R.Root, 2030/01/01, Created Starter Script
#   G.DuBuque, 2024/08/05, Updated script for Assignment 06, added SOC pattern,
#                          classes FileProcessor and IO, and their functions.
# ------------------------------------------------------------------------------------------ #
import json

# Data ---------------------------------------------------------------------- #
# Define the global Constants (GD)
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"

# Define the global Variables (GD)
students: list = []     # List of student data as dictionaries
menu_choice: str = ''   # Holds the choice made by the user.
```

Figure 1: Script Header and Program Data Layer

Processing Layer, Class and Functions

The next separation of concern, the “Processing Layer”, gathers the program’s functions specifically involving how the program works with files. In this case, reading and writing data to a JSON file. The Processing Layer starts by creating a class called *FileProcessing.* Classes are a way to encapsulate functions and variables by the class name. Right under the class definition we include a descriptive document string that includes a description of what the class is for and a change log.

Within the class we then define functions to work with the data files. We first use the *@staticmethod* decorator to indicate the following function can be used without creating an object of the class first (objects will be taught in the next assignment). We then define a function called *read_data_from_file()* that reads data from a .json file and returns that data in a list. The function includes using parameters, *file_name* and *student_data,* to pass variables as arguments into the function (to give the function the name of the file and the list variable), when it is used later in the script. All functions include a descriptive document string explaining what the function does, its parameters and what it returns, and a change log.

The code within the function is mostly the same as the starter script's section to read data from the .json file, including exception handling, with the exception that we now use a function from another class (to be defined later in the script) to print the error messages. We also define another static method, *write_data_to_file(),* in a similar way to write data to the .json file. To use a function from a class (aka method), we use a syntax in the form of *ClassName.function_name(arguments)*. See the following figures for the Processing layer of the script.

```
# Processing ------------------------------------------------------------ #
2 usages
class FileProcessor:
    """
    A collection of processing layer functions to work with JSON files.

    ChangeLog: (Who, When, What)
    G.DuBuque, 2024/08/05, Created Class
    """

    1 usage
    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """ This function reads data from a JSON file and returns the data
        in a list.

        :param file_name: Name of the JSON file
        :param student_data: List of student data
        :return: List of student data

        ChangeLog: (Who, When, What)
        G.DuBuque, 2024/08/05, Created function
        """
        try:
            file = open(file_name, "r")
            student_data = json.load(file)
            file.close()
        except FileNotFoundError as e:
            IO.output_error_messages( message: "Text file must exist before running this script!", e)
        except Exception as e:
            IO.output_error_messages( message: "There was a non-specific error!", e)
        finally:
            if not file.closed:
                file.close()
        return student_data

```

Figure 2: Processing Layer and FileProcessor Class

```
    1 usage
    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):
        """ This function writes data to a JSON file.

        :param file_name: Name of the JSON file
        :param student_data: List of student data
        :return: None

        ChangeLog: (Who, When, What)
        G.DuBuque, 2024/08/05, Created function
        G.DuBuque, 2024/08/06, Added for loop to print dictionaries in student_data
        """
        try:
            file = open(file_name, "w")
            json.dump(student_data, file)
            file.close()
        except TypeError as e:
            IO.output_error_messages( message: "Please check that the data is a valid JSON format", e)
        except Exception as e:
            IO.output_error_messages( message: "There was a non-specific error!", e)
        finally:
            if not file.closed:
                file.close()
            print("The following data was saved to the file!")
            # Changed print format to show data as is (dictionaries in json format) (GD)
            for student in student_data:
                print(student)
```

Figure 3: Processing Layer and FileProcessor Class continued

Presentation Layer, Class and Functions

The next separation of concern, the "Presentation Layer", gathers the program's functions to display and get data from the program user, or the program's Input/Output (IO). We define the class *IO,* and the following static methods: *output_error_messages*, *output_menu*, *input_menu_choice, output_student_courses, input_student_data*. As before, all functions include descriptive document strings, parameters and returns as necessary, and the code is mostly the same as the starter file with the exception of now using the *IO.output_error_messages()* method to print error messages. Additionally, I added the default value of *None* to the *output_error_messages* parameter *message* to make the function more robust so it can still be used without providing a *message* argument. See the following figures for the Presentation layer of the script.

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

    ChangeLog: (Who, When, What)
    G.DuBuque, 2024/08/05, Created Class
    """

    7 usages
    @staticmethod
    def output_error_messages(message: str = None, error: Exception = None):
        """ This function displays a given custom error message and the
        built-in error messages of a given Exception object to the user.

        :param message: Custom error message
        :param error: Exception object
        :return: None

        ChangeLog: (Who, When, What)
        G.DuBuque, 2024/08/05, Created function, added default of None to
        message parameter to add robustness to function
        """
        if message is None:  # In case of not providing a custom message
            print("There was an error!", "\n")
        else:
            print(message, "\n")
        if error is not None:  # Print built-in error messages
            print("-- Technical Error Message -- ")
            print(error, error.__doc__, type(error), sep='\n')

    1 usage
    @staticmethod
    def output_menu(menu: str):
        """ This function displays the menu of choices to the user.

        :param menu: Menu to display
        :return: None

        ChangeLog: (Who, When, What)
        G.DuBuque, 2024/08/05, Created function
        """
        print(menu)

```

Figure 4: Presentation Layer and IO Class

```
    1 usage
    @staticmethod
    def input_menu_choice():
        """ This function returns the menu of choice of the user as a string.

        :param: None
        :return: String with the user's choice

        ChangeLog: (Who, When, What)
        G.DuBuque, 2024/08/05, Created function
        """
        choice = "0"
        try:
            choice = input("Enter your menu choice number: ")
            if choice not in ("1", "2", "3", "4"):
                raise Exception("Please choose only 1, 2, 3 or 4")
        except Exception as e:
            IO.output_error_messages(error=e)  # Print error messages

        return choice

    1 usage
    @staticmethod
    def output_student_courses(student_data: list):
        """ This function displays the complete list of registration data
        to the user in a formatted string.

        :param student_data: List of student data
        :return: None

        ChangeLog: (Who, When, What)
        G.DuBuque, 2024/08/05, Created function
        """
        print("-" * 50)
        for student in student_data:
            print(f"Student {student["FirstName"]} "
                  f"{student["LastName"]} is enrolled in {student["CourseName"]}")
        print("-" * 50)

```

Figure 5: Presentation Layer and IO Class continued

```python
    1 usage
    @staticmethod
    def input_student_data(student_data: list):
        """ This function gets the student's first name, last name and
        course name from the user, and updates and returns the list with the
        new data.

        :param student_data: List of student data
        :return: Updated list of student data

        ChangeLog: (Who, When, What)
        G.DuBuque, 2024/08/05, Created function
        """
        try:
            student_first_name = input("Enter the student's first name: ")
            if not student_first_name.isalpha():
                raise ValueError("The first 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}
            student_data.append(student)
            print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
        except ValueError as e:
            IO.output_error_messages( message: "The data entered is not in the correct format!", e)
        except Exception as e:
            IO.output_error_messages(error=e)
        return student_data

```

Figure 6: Presentation Layer and IO Class continued

Main Body of Script

After all the classes and methods are defined in their areas of concern, we start the main body of the script where we call the functions as needed to run our program. The order of operations and flow of the main program block is the same as the starter file, but the syntax is now more intuitive, concise and easier to read. For functions that return values we assign them to the appropriate global variable, and for functions that have parameters we pass the appropriate global variables as arguments. See **Figure 7** for the main body of the script.

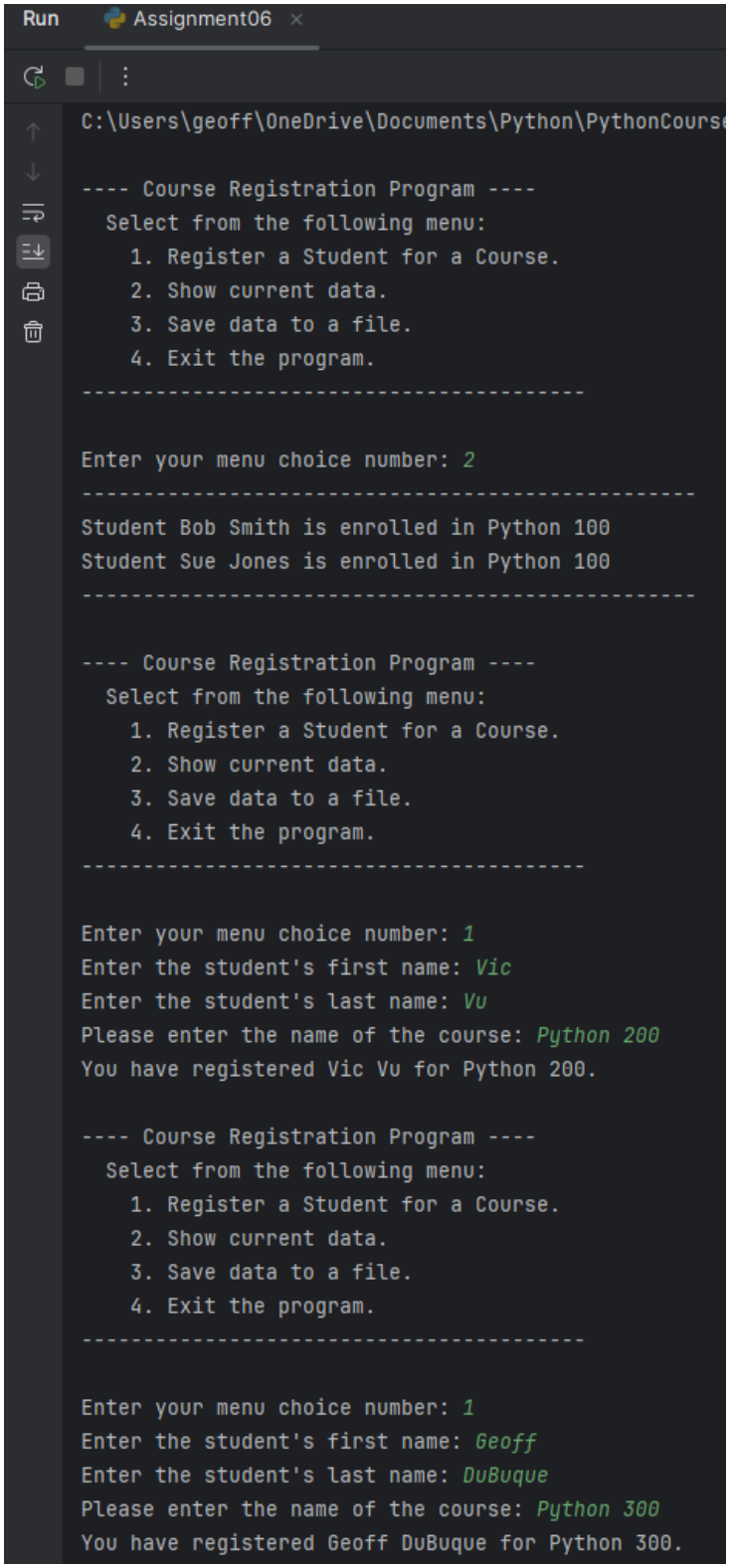
```python
# Main body of script ------------------------------------------------------ #
# Get current data
students = FileProcessor.read_data_from_file(FILE_NAME, students)

# Present and Process the data
while True:
    IO.output_menu(MENU)  # Present the menu of choices

    menu_choice = IO.input_menu_choice()  # Get menu choice

    if menu_choice == "1":  # Input user data
        students = IO.input_student_data(students)
        continue

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

    elif menu_choice == "3":  # Save the data to a file
        FileProcessor.write_data_to_file(FILE_NAME, students)
        continue

    elif menu_choice == "4":  # Stop the program
        break  # out of the loop

print("Program Ended")
```

Figure 7: Menu Body of Script

Testing the Program

I tested the program by first choosing menu choice 2 to display the existing data in the file and make sure it is displayed in the correct format. Then I use menu choice 1 a couple times to add some new data. I used choice 2 again to make sure the new data was added. I then use choice 3 to save the data and make sure it is displayed correctly. I ended the program with choice 4.

I tested the error handling by changing the file name in the FILE_NAME variable and entering numbers in the first and last name inputs to make sure the corresponding errors are displayed. The program is run in both PyCharm and Windows PowerShell. The results of the program running correctly are shown on the next pages.



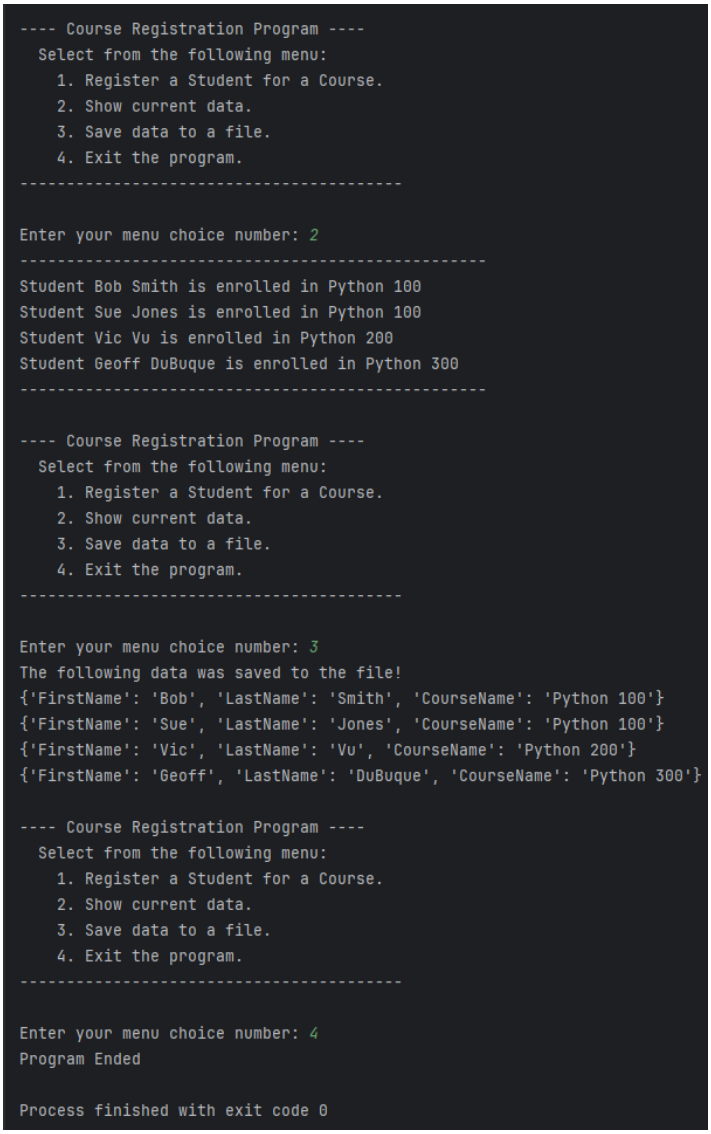
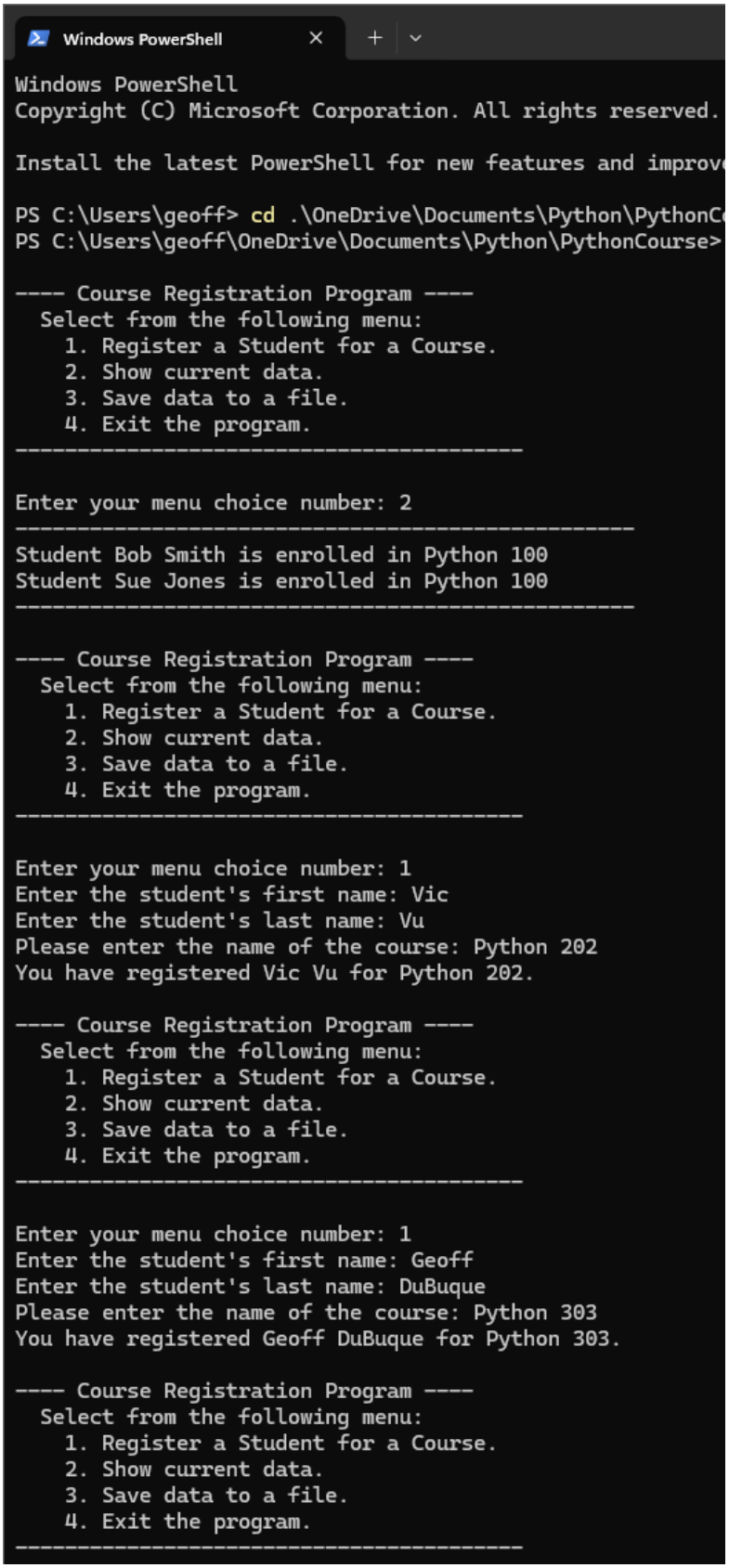


Figure 8: Program Ran in PyCharm



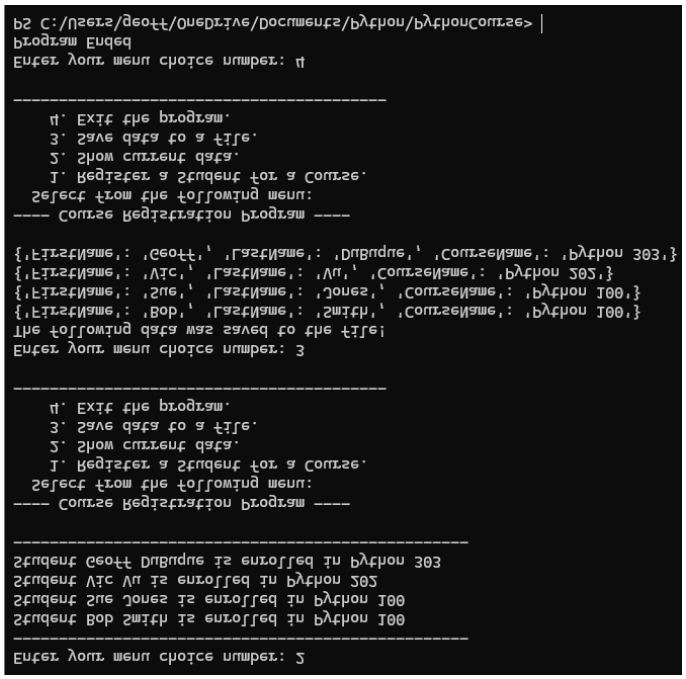
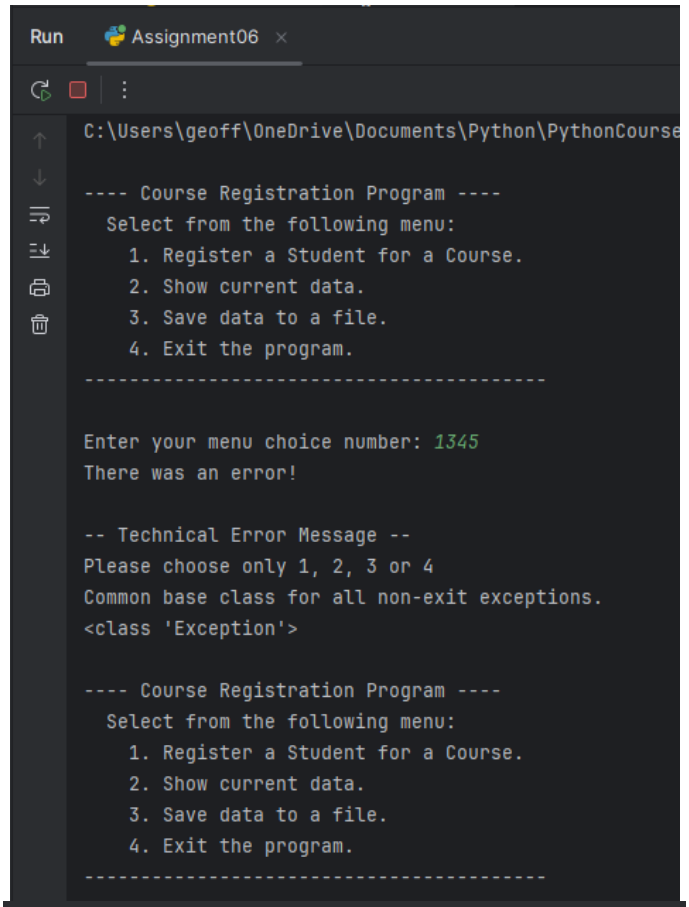
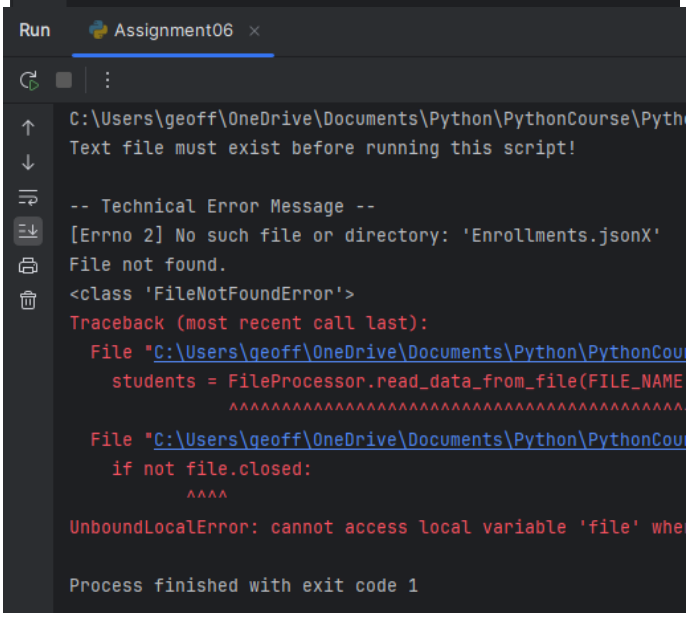


Figure 9: Program Ran in PowerShell





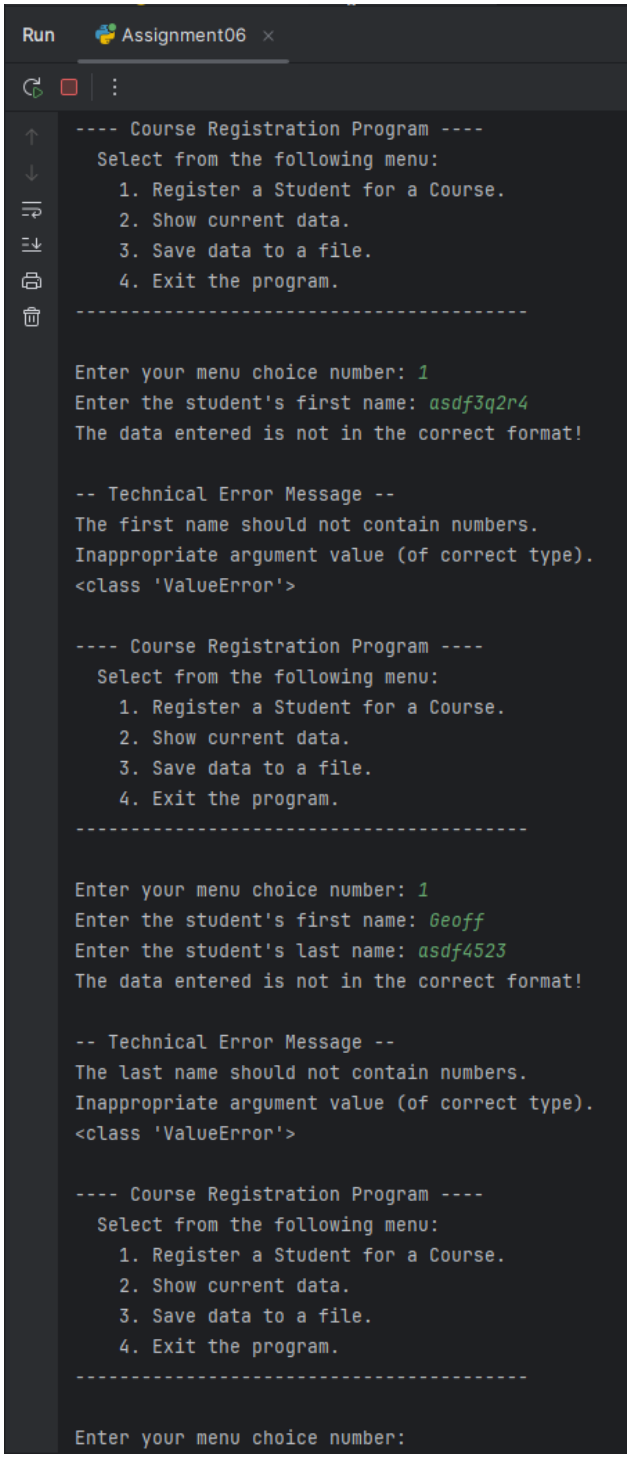


Figure 10: Error Handling in PyCharm

Summary

This programming assignment introduced common and powerful ways to refactor our code to add many benefits, such as readability, modularity, reusability, encapsulation, flexibility, maintainability, and ease of collaboration. We used 3 techniques to reorganize our code: separation of concerns pattern, classes and functions. We also learned how to use classes with static methods and parameters, arguments, and return values. I'm looking forward to how we will use objects of classes in our next assignment.

References

Randal Root. 2024. Mod06-Notes.docx. IT FDN 110 A Su 24: Foundations of Programming: Python. University of Washington, Professional & Continuing Education

Randal Root. 2024. Mod06-Assignment.docx. IT FDN 110 A Su 24: Foundations of Programming: Python. University of Washington, Professional & Continuing Education

Randal Root. 2024. Assignment06-Starter.py. IT FDN 110 A Su 24: Foundations of Programming: Python. University of Washington, Professional & Continuing Education

Randal Root. 2024. Mod06-Lab03-WorkingWithClassesAndSoC.py. IT FDN 110 A Su 24: Foundations of Programming: Python. University of Washington, Professional & Continuing Education