Monisha Grover
November 28, 2024
IT FDN 110B
Assignment 07
https://github.com/MonishaGrover/IntroToProg-Python-Mod07

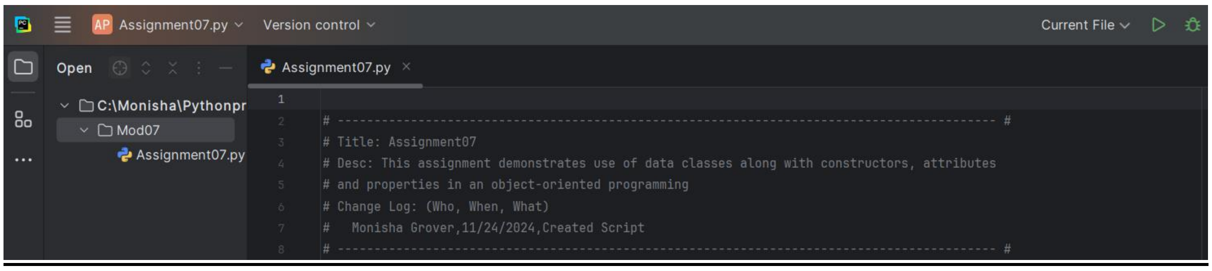
Writing an object-oriented program using constructors, attributes and properties in Data Classes

Introduction

This script demonstrates the use of Python programming concepts such as encapsulation and the separation of concerns design pattern. It highlights the use of constructors, attributes and properties which are essential elements of Object-Oriented Programming. The program simulates a course registration system where users can register students for courses, view registered data, save the data to a file, and exit the application.

Code Documentation

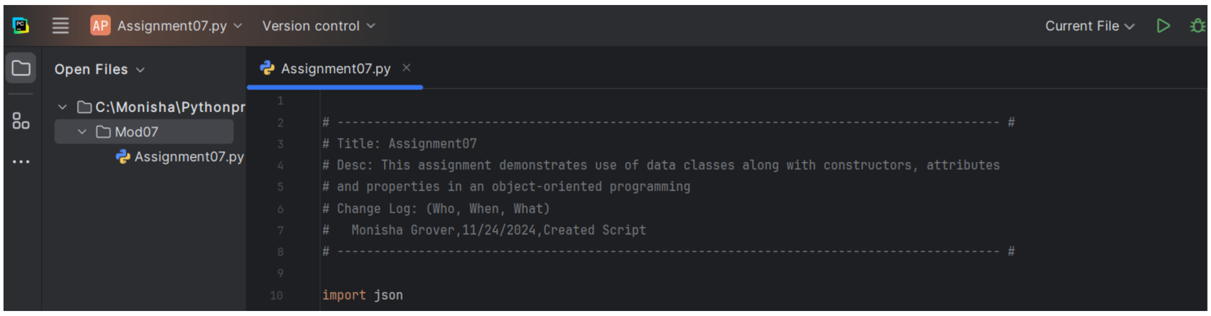
- **Script Header:** I updated the script header with the title, description and information on the creation of document. This helps in the documentation and maintenance of the program in the future iterations.



```
# ---------------------------------------------------------------------------------------- #
# Title: Assignment07
# Desc: This assignment demonstrates use of data classes along with constructors, attributes
# and properties in an object-oriented programming
# Change Log: (Who, When, What)
#   Monisha Grover,11/24/2024,Created Script
# ---------------------------------------------------------------------------------------- #
```

Figure01: Script header

- **Imports:** The json module is imported to handle the serialization and deserialization of student data. This ensures that data can be saved in a structured format to a file and reloaded when needed.



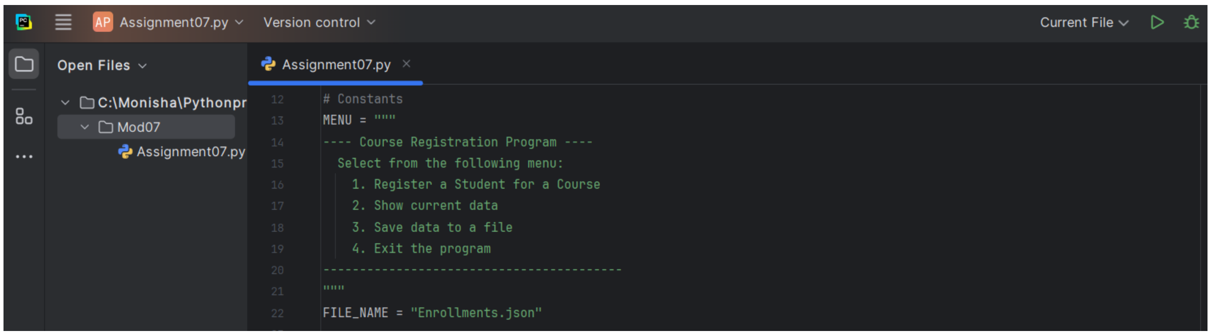
```
# ---------------------------------------------------------------------------------------- #
# Title: Assignment07
# Desc: This assignment demonstrates use of data classes along with constructors, attributes
# and properties in an object-oriented programming
# Change Log: (Who, When, What)
#   Monisha Grover,11/24/2024,Created Script
# ---------------------------------------------------------------------------------------- #

import json
```

Figure02: Importing Json

- **Declaring the constants and variables:** Using the information provided in the assignment, I wrote all the constants in Uppercase letters separated with underscores and assigned them data type (str/float). The following constants were used:

  MENU: A string constant that displays the main menu options available in the program.

FILE_NAME: The name of the file (Enrollments.json) where the student enrollment data will be stored or retrieved.

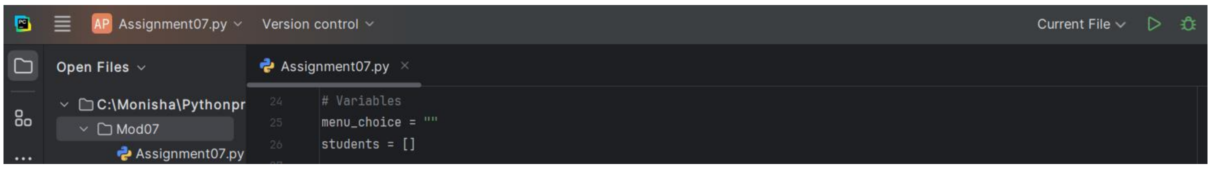


```
# Constants
MENU = """
---- Course Registration Program ----
  Select from the following menu:
    1. Register a Student for a Course
    2. Show current data
    3. Save data to a file
    4. Exit the program
-----------------------------------------
"""
FILE_NAME = "Enrollments.json"
```

Figure03: Declaring constants

Next, I **declared the Variables and initialized them to empty string** by putting **str()** after the assignment (=) operator. The following variables were used:

menu_choice: Stores the user's menu selection input.
students: A **list of dictionaries** that stores data for all enrolled students in a nested list format.

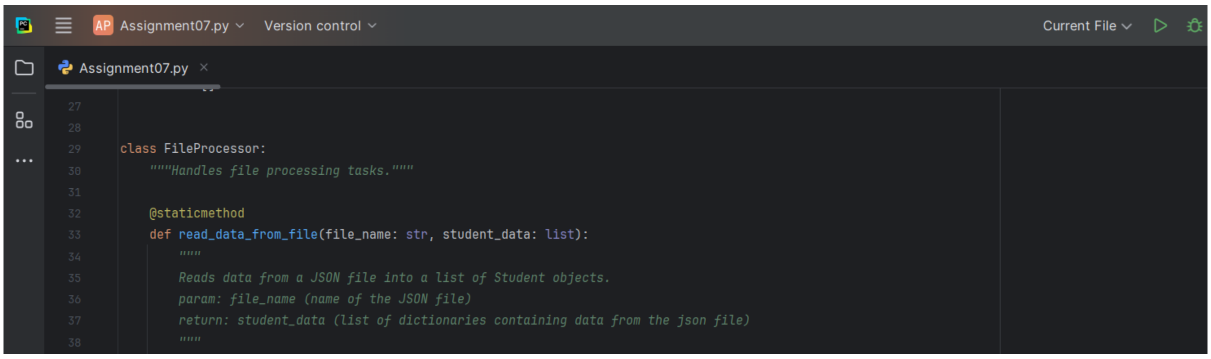


```
# Variables
menu_choice = ""
students = []
```

Figure04: Declaring variables

- **Class: FileProcessor**

This class encapsulates file handling operations, adhering to the *separation of concerns* principle. Each class and method is followed by a docstring which provides a concise description of the class's purpose. For Example:



```
class FileProcessor:
    """Handles file processing tasks."""

    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """
        Reads data from a JSON file into a list of Student objects.
        param: file_name (name of the JSON file)
        return: student_data (list of dictionaries containing data from the json file)
        """
```

Figure05: Class FileProcessor

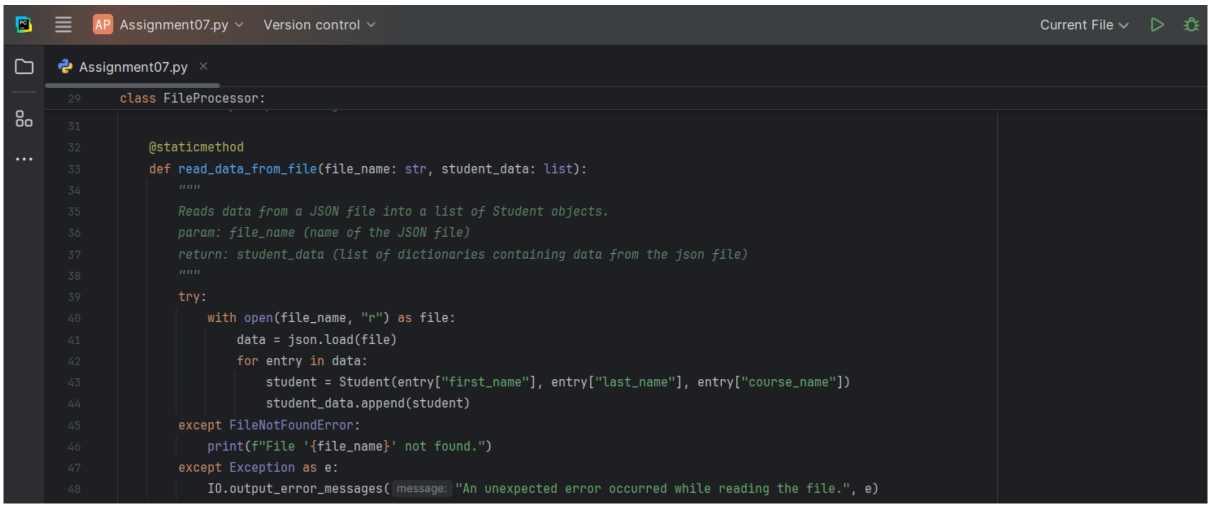
The **@staticmethod decorator** indicates that the methods in the class do not require access to the class instance (self) or class-level data (cls). These methods act as utility functions that can be called using FileProcessor.method_name() without creating an instance of the class. It provides the following static methods:

Method: read_data_from_file

Reads JSON data from a file and converts it into **Student** objects, appending them to the **student_data** list. This function does not explicitly return a value; instead, it modifies the **student_data** list in place.

Uses error handling to catch file-related exceptions, such as FileNotFoundError.

with open(file_name, "r"): Ensures the file is properly opened and closed automatically.
json.load(file): Reads the file's content and parses it into a Python data structure (a list of dictionaries).
for entry in data: Loops through each dictionary (representing a student's data) in the JSON file.
Student(entry["first_name"], ...): Creates a new Student object using the first_name, last_name, and course_name values from the dictionary.
student_data.append(student): Adds the Student object to the student_data list.

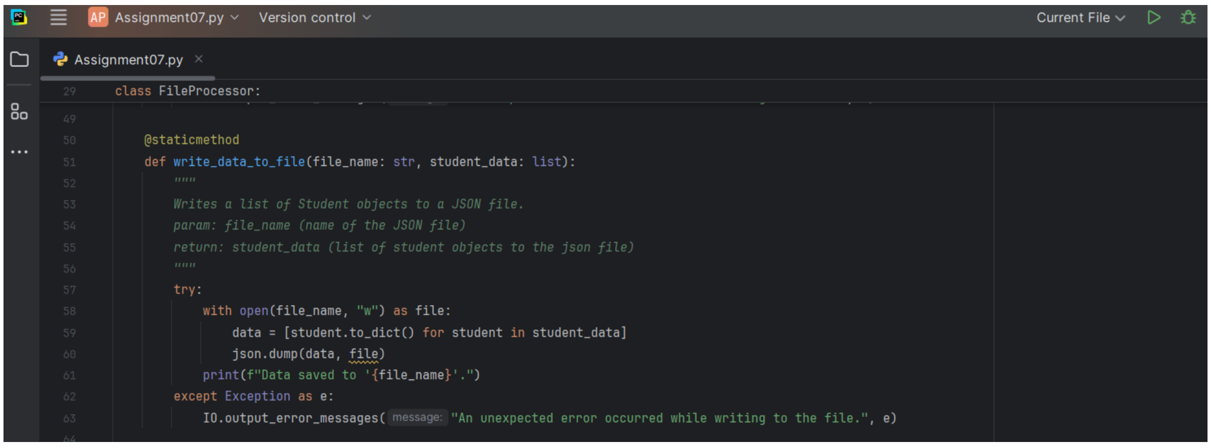


```
class FileProcessor:

    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """
        Reads data from a JSON file into a list of Student objects.
        param: file_name (name of the JSON file)
        return: student_data (list of dictionaries containing data from the json file)
        """
        try:
            with open(file_name, "r") as file:
                data = json.load(file)
                for entry in data:
                    student = Student(entry["first_name"], entry["last_name"], entry["course_name"])
                    student_data.append(student)
        except FileNotFoundError:
            print(f"File '{file_name}' not found.")
        except Exception as e:
            IO.output_error_messages( message: "An unexpected error occurred while reading the file.", e)
```

Figure06: Method: read_data_from_file

Method: write_data_to_file

Writes the **student_data** list (containing **student** objects) to a JSON file.
with open(file_name, "w"): Opens the file in write mode ("w") to overwrite its contents.
data = [student.to_dict() for student in student_data]: Converts each Student object into a dictionary using the **to_dict() method**. It is a custom method commonly used in classes to convert an instance of the class into a Python dictionary.
json.dump(data, file): Serializes the list of dictionaries (data) into JSON format and writes it to the file.
print: Confirms that the data was successfully saved.



```
class FileProcessor:

    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):
        """
        Writes a list of Student objects to a JSON file.
        param: file_name (name of the JSON file)
        return: student_data (list of student objects to the json file)
        """
        try:
            with open(file_name, "w") as file:
                data = [student.to_dict() for student in student_data]
                json.dump(data, file)
            print(f"Data saved to '{file_name}'.")
        except Exception as e:
            IO.output_error_messages( message: "An unexpected error occurred while writing to the file.", e)
```

Figure07: Method: write_data_to_file

- **Class: IO**

  The IO class handles user interaction and input/output operations. It is designed to keep user-interface logic separate from file and core processing. Each class and method is followed by a docstring which provides a concise description of the class's purpose. The @staticmethod decorator indicates that the methods in the class do not require access to the class instance (self) or class-level data (cls). These methods act as utility functions that can be called using IO.method_name() without creating an instance of the class. It provides the following static methods:

  Method: output_error_messages

  Displays error messages along with optional technical details.

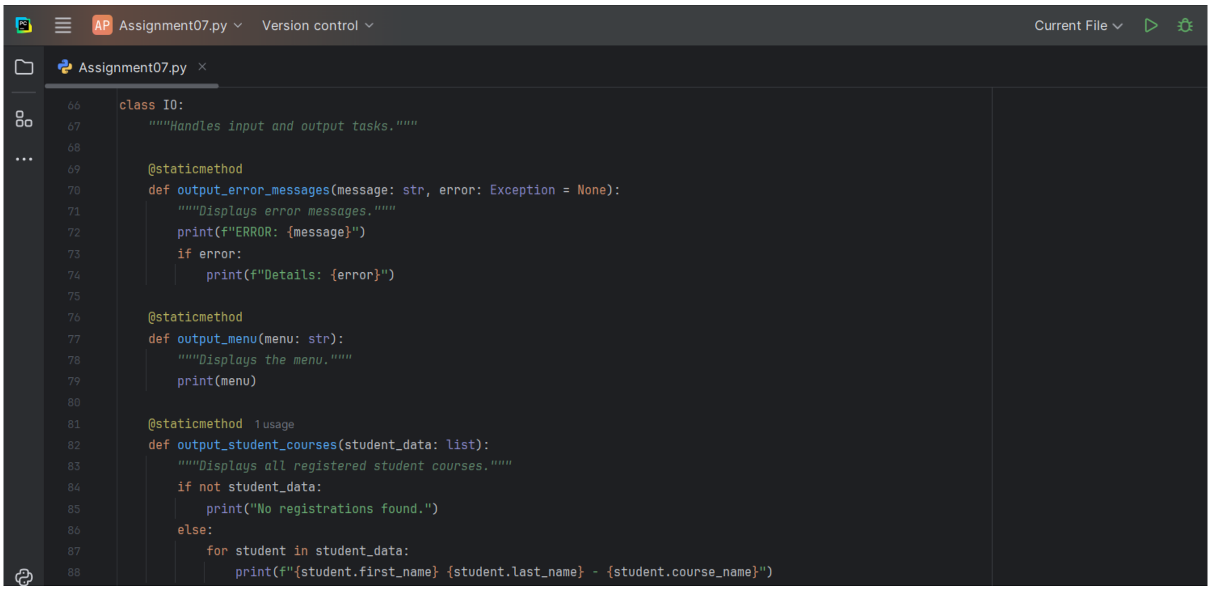
  Method: output_menu

  Displays the main menu to the user.

  Method: output_student_courses

  Displays all registered students and their courses. If no data exists, prompts the user to register students first. **student_data** is a list of Student objects.

  if not student_data: Checks if the list is empty. If it is, a message is printed indicating no data.
  for student in student_data: Iterates over the list.
  print(f"{student.first_name} ..."): Outputs each student's name and their course.



```python
class IO:
    """Handles input and output tasks."""

    @staticmethod
    def output_error_messages(message: str, error: Exception = None):
        """Displays error messages."""
        print(f"ERROR: {message}")
        if error:
            print(f"Details: {error}")

    @staticmethod
    def output_menu(menu: str):
        """Displays the menu."""
        print(menu)

    @staticmethod
    def output_student_courses(student_data: list):
        """Displays all registered student courses."""
        if not student_data:
            print("No registrations found.")
        else:
            for student in student_data:
                print(f"{student.first_name} {student.last_name} - {student.course_name}")
```

Figure08: class IO: Output

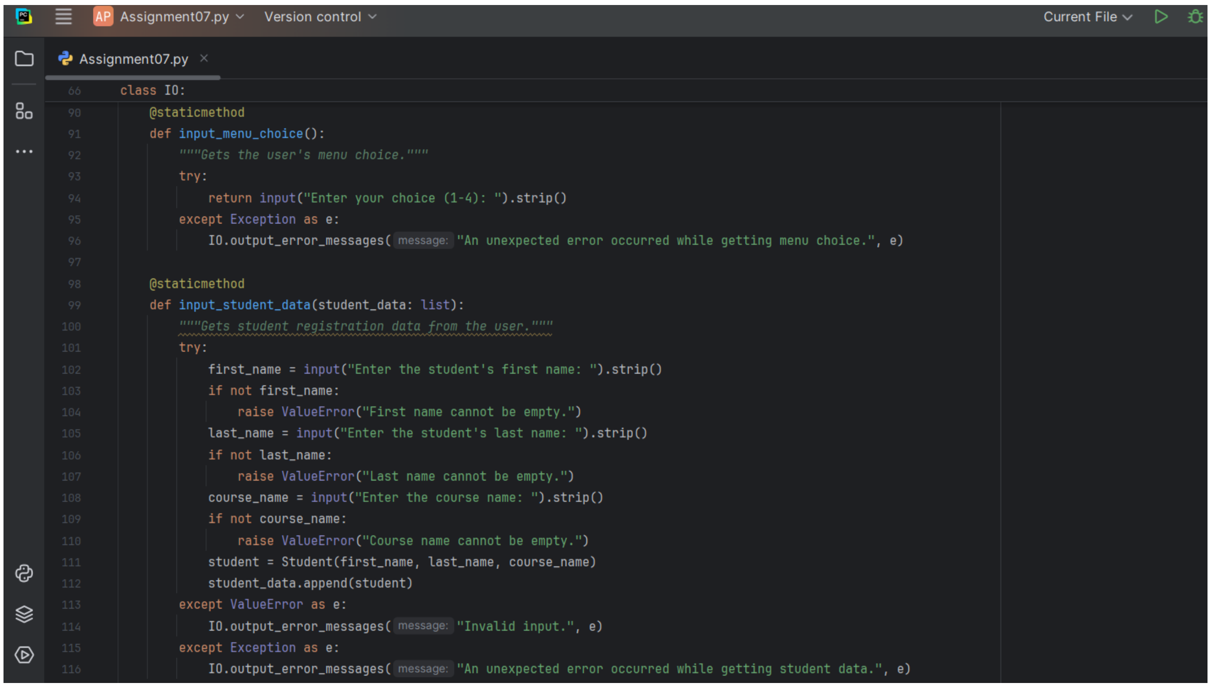
  Method: input_menu_choice

  Prompts the user for a menu selection and validates the input.

  Method: input_student_data

  Prompts the user to input student details and appends them to the students list. Each field is checked to ensure it is not empty. If empty, a ValueError is raised with an appropriate message. Handles invalid input through exceptions.

student = Student(first_name, ...): Creates a Student object using the provided inputs.
student_data.append(student): Adds the new student to the list.



```python
class IO:
    @staticmethod
    def input_menu_choice():
        """Gets the user's menu choice."""
        try:
            return input("Enter your choice (1-4): ").strip()
        except Exception as e:
            IO.output_error_messages( message: "An unexpected error occurred while getting menu choice.", e)

    @staticmethod
    def input_student_data(student_data: list):
        """Gets student registration data from the user."""
        try:
            first_name = input("Enter the student's first name: ").strip()
            if not first_name:
                raise ValueError("First name cannot be empty.")
            last_name = input("Enter the student's last name: ").strip()
            if not last_name:
                raise ValueError("Last name cannot be empty.")
            course_name = input("Enter the course name: ").strip()
            if not course_name:
                raise ValueError("Course name cannot be empty.")
            student = Student(first_name, last_name, course_name)
            student_data.append(student)
        except ValueError as e:
            IO.output_error_messages( message: "Invalid input.", e)
        except Exception as e:
            IO.output_error_messages( message: "An unexpected error occurred while getting student data.", e)
```

Figure09: class IO: Input

- <u>**Class: Person**</u>

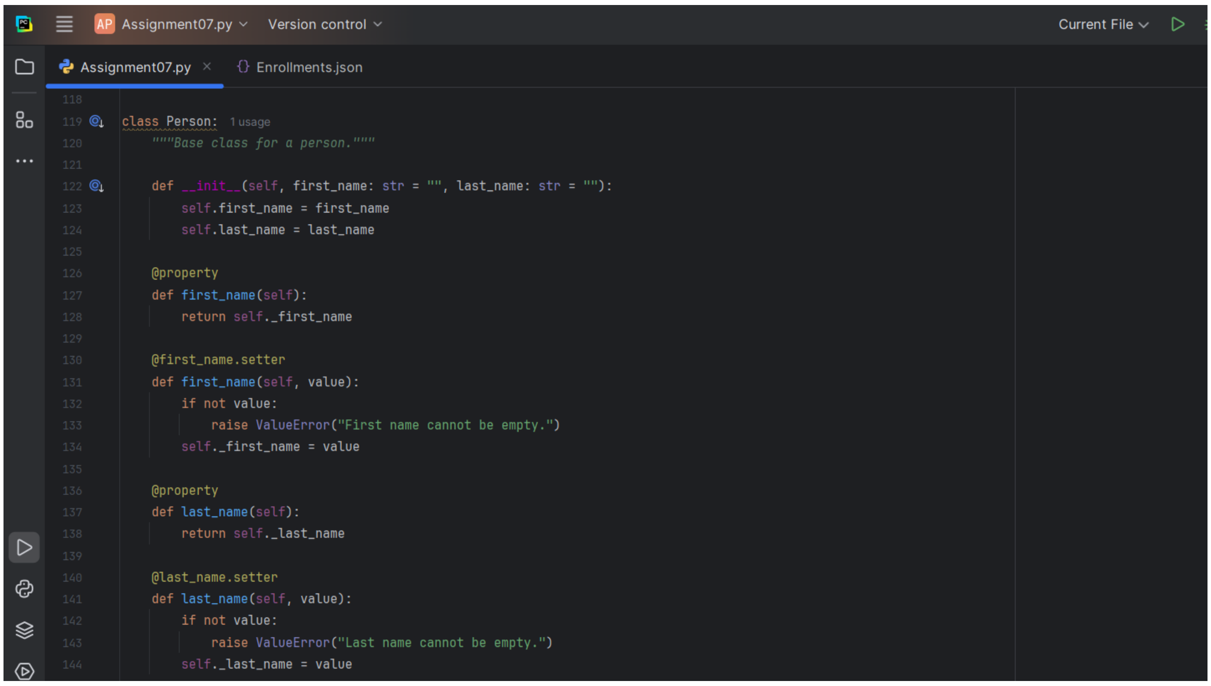
Represents a base class for a person with basic attributes like first name and last name. It implements the principle of encapsulation using private variables and properties. Private variables (_first_name and _last_name) store the values of the attributes and ensure that all modifications go through the setter methods, which enforce validation.

Method: **__init__**
This constructor method initializes the object with a first name and a last name.

self.first_name = first_name: Sets the first_name property using the setter method for validation.
self.last_name = last_name: Sets the last_name property using the setter method for validation.
The keyword "self" refer to data or functions found in an object instance. When the script runs, the class's code immediately loads into memory and then waits to be used, either directly or through an object instance. While the class's code only loads into memory once, it can have many object instances of a class, each representing a "copy"of the classes code.

Properties
Properties are functions designed to manage attribute data. Typically, for each attribute two types of properties can be created: "getter" and "setter"
@property
def first_name(self):
@property decorator indicates <u>"getter" function</u> for the attribute first_name.
return self._first_name: The return statement inside the first_name property retrieves the value of the private attribute _first_name.

@first_name.setter
def first_name(self, value):
<u>Setter property function</u> allows to add validation and error handling.
@first_name.setter allows us to set the value of first_name.
if not value: Checks if the input is empty or None and raises a ValueError with the message "First name cannot be empty." if validation fails.
self._first_name = value: Assigns the value to the private _first_name variable if validation passes.



```
class Person:
    """Base class for a person."""

    def __init__(self, first_name: str = "", last_name: str = ""):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not value:
            raise ValueError("First name cannot be empty.")
        self._first_name = value

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if not value:
            raise ValueError("Last name cannot be empty.")
        self._last_name = value
```

Figure10: class Person

- <u>**Class: Student (Person)**</u>
  It defines a new class named Student, which **inherits** from the Person class. This means Student will have access to all methods and properties of the Person class, unless overridden.

  Method: **__init__**
  The constructor method initializes an instance of Student class. first_name and last_name are the parameters the constructor accepts. Each has a default value of an empty string.
  super().__init__(first_name, last_name)
  Calls the constructor of the Person class (the parent class) to initialize first_name and last_name.
  self.course_name = course_name
  Sets the course_name attribute for the instance.

Getter Method for course_name

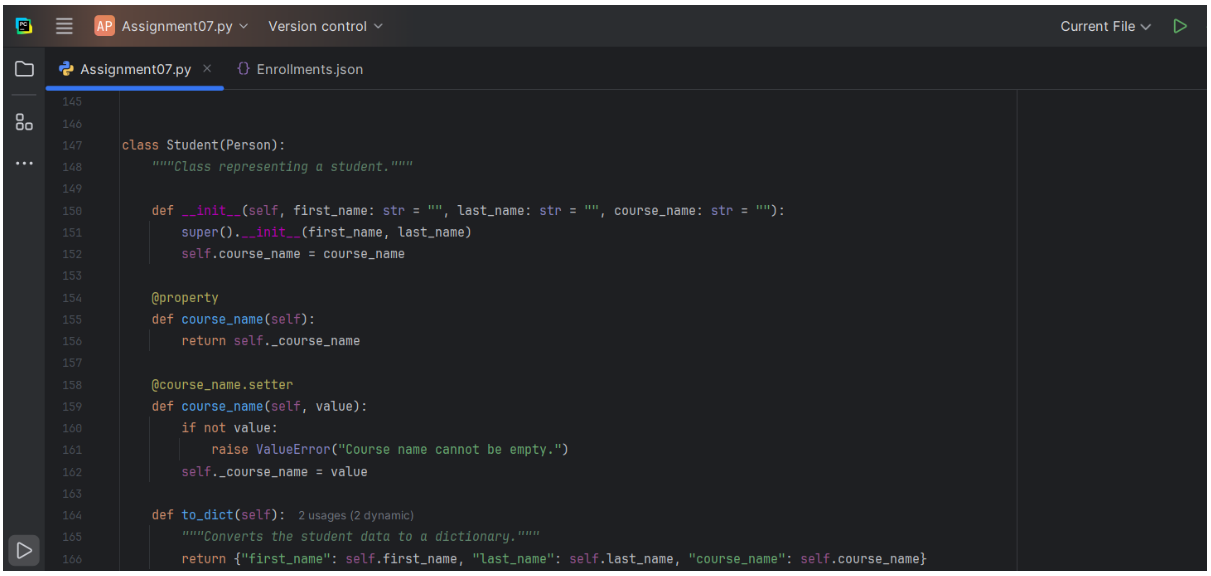
@property
Marks the following method as a getter for course_name. This allows you to access course_name as if it were a simple attribute, e.g., student.course_name.
def course_name(self):
Defines the getter method for course_name.

return self._course_name
Returns the value of the private attribute _course_name. The underscore indicates that _course_name is meant to be "private" (a convention in Python).

Setter Method for course_name

@course_name.setter
Marks the following method as a **setter** for course_name. This allows you to assign a value to course_name, e.g., student.course_name = "Math".
def course_name(self, value):
Defines the setter method for course_name.
if not value:
Checks if value is empty or evaluates to False.
raise ValueError("Course name cannot be empty.")
Raises a ValueError if the input is invalid. This ensures that course_name cannot be set to an empty value.
self._course_name = value
Assigns the validated value to the private attribute _course_name.

Method: to_dict
def to_dict(self):
Defines a method to convert the Student object into a dictionary.
return {...}
Creates and returns a dictionary with the student's data:
first_name: The student's first name (inherited from Person).
last_name: The student's last name (inherited from Person).
course_name: The student's course name (defined in Student).



```python
class Student(Person):
    """Class representing a student."""

    def __init__(self, first_name: str = "", last_name: str = "", course_name: str = ""):
        super().__init__(first_name, last_name)
        self.course_name = course_name

    @property
    def course_name(self):
        return self._course_name

    @course_name.setter
    def course_name(self, value):
        if not value:
            raise ValueError("Course name cannot be empty.")
        self._course_name = value

    def to_dict(self):
        """Converts the student data to a dictionary."""
        return {"first_name": self.first_name, "last_name": self.last_name, "course_name": self.course_name}
```

Figure10: class student(Person)

- <u>**Main Program**</u>

  The main block ensures that the program executes only when run as a script.

  if __name__ == "__main__":

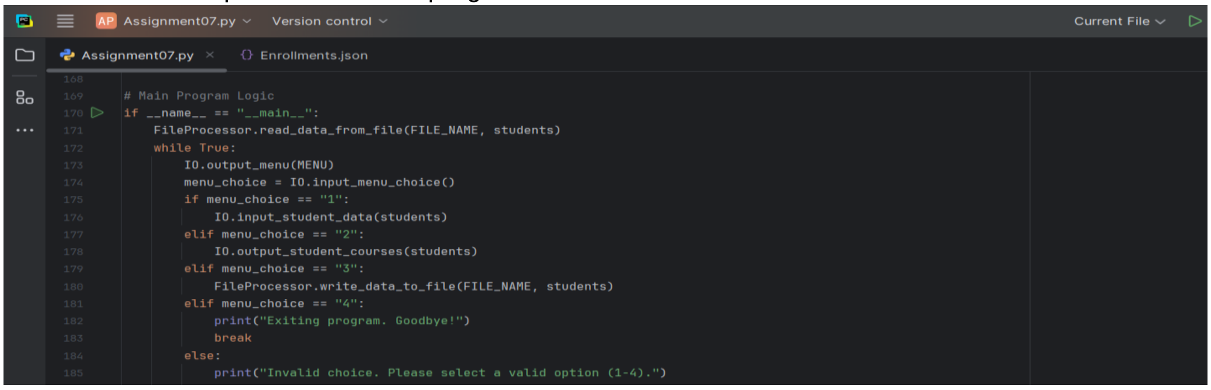
  This ensures that the code block below it will only run **if the script is executed directly** (not imported as a module in another script).

The special variable __name__ is set to "__main__" when the script is run directly. If the script is imported, __name__ will instead be set to the name of the module.

The FileProcessor.read_data_from_file method loads pre-existing data from the JSON file. Calls the read_data_from_file method of the FileProcessor class.

A while loop continuously displays the menu and handles user selections:

1. Option 1: Registers a new student using IO.input_student_data.
2. Option 2: Displays registered students using IO.output_student_courses.
3. Option 3: Saves current data to a file using FileProcessor.write_data_to_file.
4. Option 4: Exits the program.



```python
# Main Program Logic
if __name__ == "__main__":
    FileProcessor.read_data_from_file(FILE_NAME, students)
    while True:
        IO.output_menu(MENU)
        menu_choice = IO.input_menu_choice()
        if menu_choice == "1":
            IO.input_student_data(students)
        elif menu_choice == "2":
            IO.output_student_courses(students)
        elif menu_choice == "3":
            FileProcessor.write_data_to_file(FILE_NAME, students)
        elif menu_choice == "4":
            print("Exiting program. Goodbye!")
            break
        else:
            print("Invalid choice. Please select a valid option (1-4).")
```

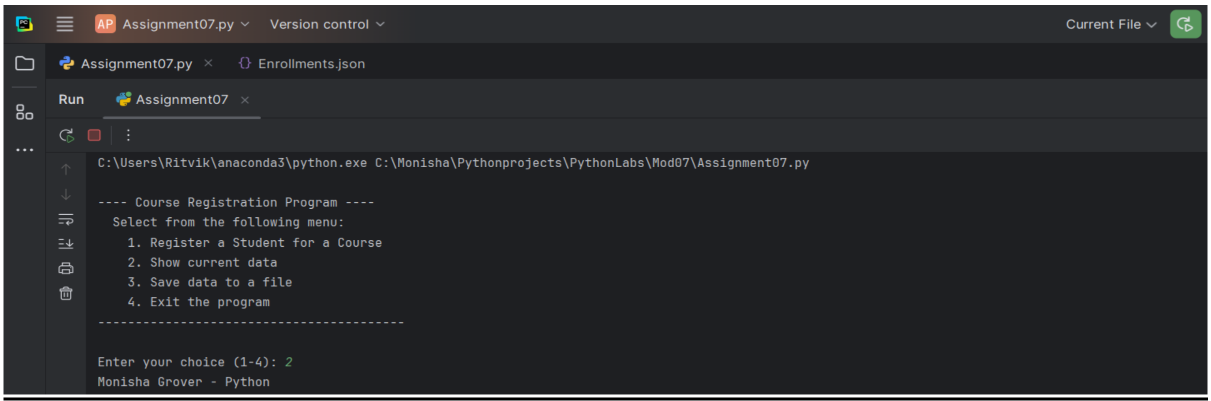
Figure11: Main Program

Testing

The final output was as desired for all four menu choices.

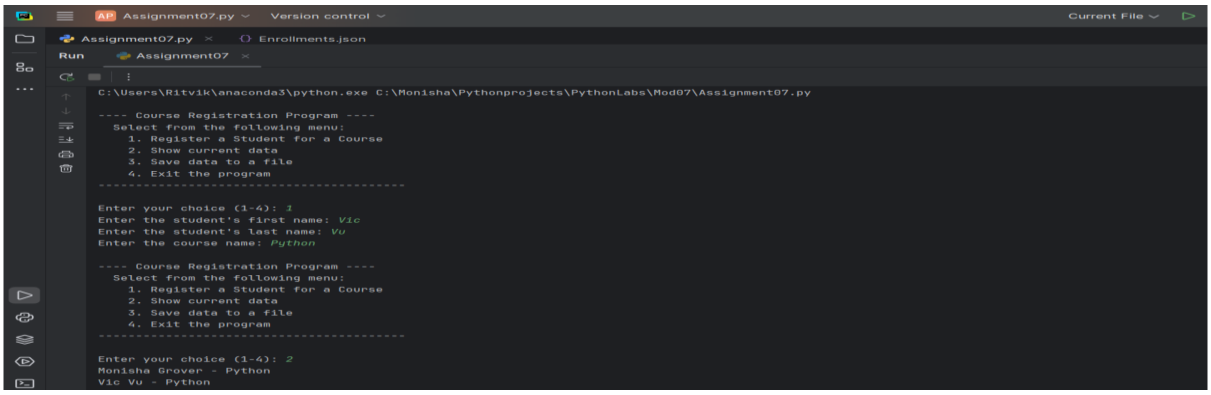
PyCharm Outputs

I began by ensuring that previous data is still available. I chose the menu choice 2 and got the data for previously enrolled students.



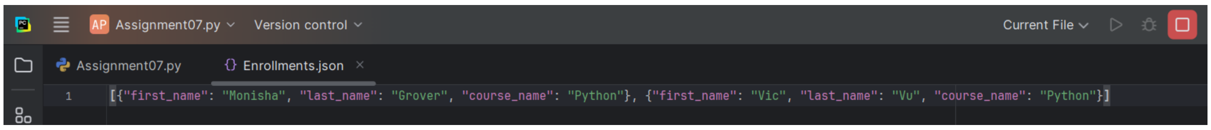
```
C:\Users\Ritvik\anaconda3\python.exe C:\Monisha\Pythonprojects\PythonLabs\Mod07\Assignment07.py

---- 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 choice (1-4): 2
Monisha Grover - Python
```

Figure12 Retrieving previous data



```
C:\Users\Ritvik\anaconda3\python.exe C:\Monisha\Pythonprojects\PythonLabs\Mod07\Assignment07.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 choice (1-4): 1
Enter the student's first name: Vic
Enter the student's last name: Vu
Enter the course name: Python

---- 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 choice (1-4): 2
Monisha Grover - Python
Vic Vu - Python
```

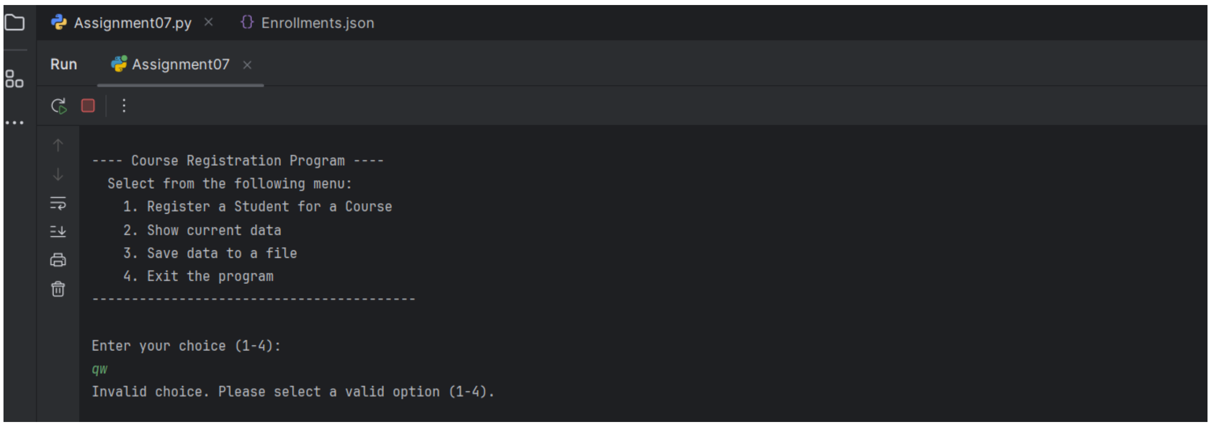
Figure13 Output for adding and appending data



```
[{"first_name": "Monisha", "last_name": "Grover", "course_name": "Python"}, {"first_name": "Vic", "last_name": "Vu", "course_name": "Python"}]
```

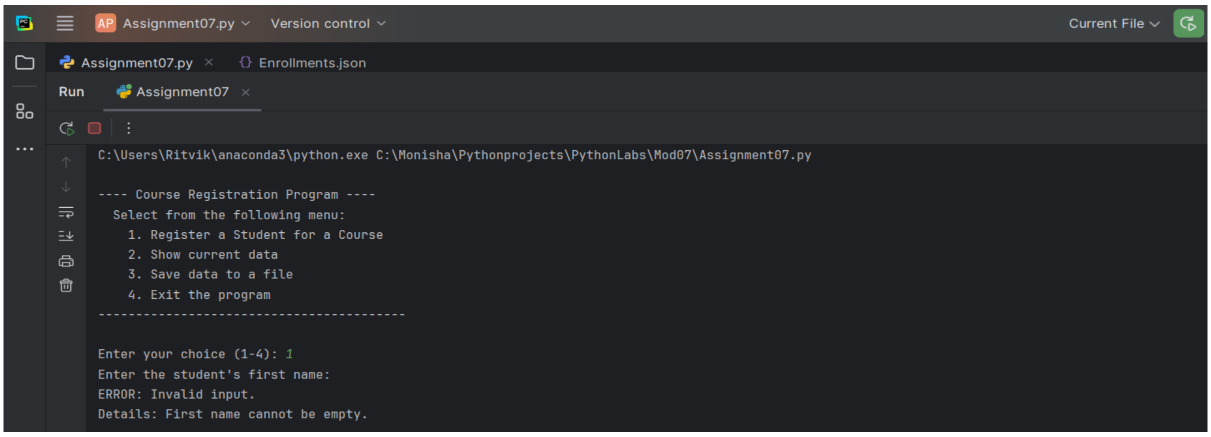
Figure14 Output: Enrollments.json file

Error handling



```
---- 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 choice (1-4):
qw
Invalid choice. Please select a valid option (1-4).
```

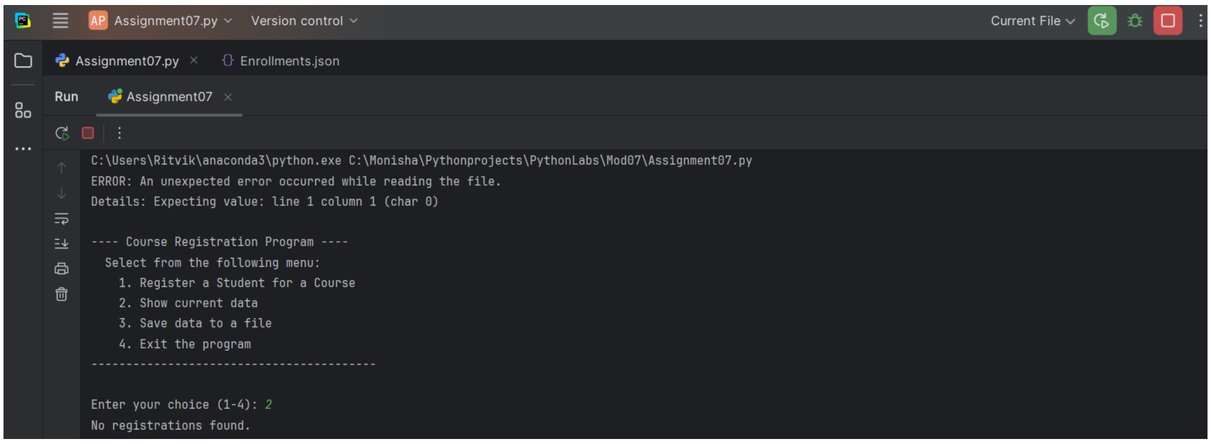
Figure15 Invalid choice error handling



```
C:\Users\Ritvik\anaconda3\python.exe C:\Monisha\Pythonprojects\PythonLabs\Mod07\Assignment07.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 choice (1-4): 1
Enter the student's first name:
ERROR: Invalid input.
Details: First name cannot be empty.
```

Figure16 Invalid input error handling

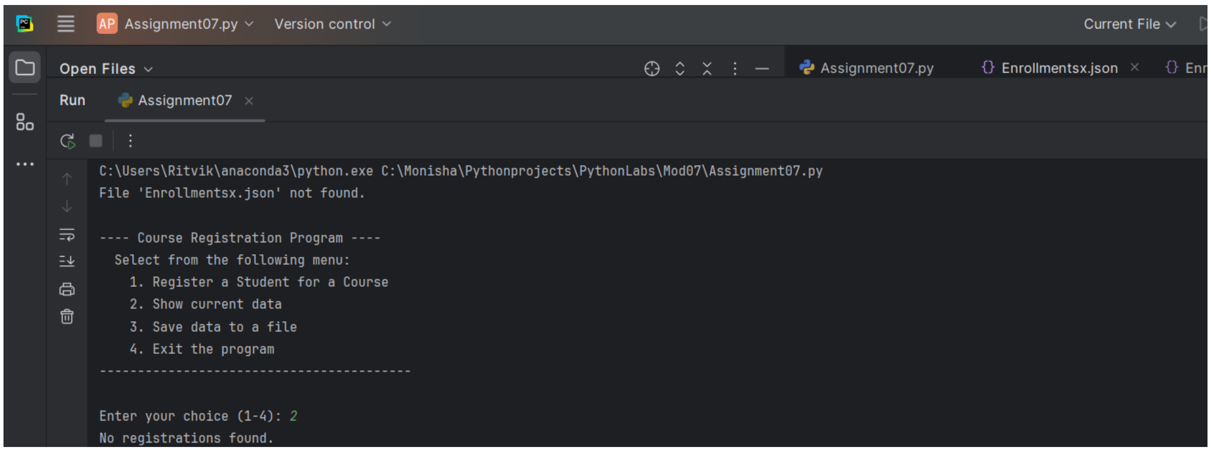
I deleted the previous data and executed the script. I got the “No registrations found” error message as expected.



```
C:\Users\Ritvik\anaconda3\python.exe C:\Monisha\Pythonprojects\PythonLabs\Mod07\Assignment07.py
ERROR: An unexpected error occurred while reading the file.
Details: Expecting value: line 1 column 1 (char 0)

---- 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 choice (1-4): 2
No registrations found.
```

Figure17 “No registrations found” error handling

I changed the json file name in the script and consequently it threw “File not found” error when executed. Since there was no file created yet the menu choice 2 correctly gave “No file registrations found”



```
C:\Users\Ritvik\anaconda3\python.exe C:\Monisha\Pythonprojects\PythonLabs\Mod07\Assignment07.py
File 'Enrollmentsx.json' not found.

---- 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 choice (1-4): 2
No registrations found.
```

Figure18 “File not found” and “No registrations found” error handling

Summary

As we keep on adding functionalities to a program, the code tends to get longer and complex. Using the concepts in Module07, I was able to add the concept of encapsulation in addition to the functions and data validation feature. Encapsulation is the act of hiding the internal details and implementation of a class by bundling the data (attributes or properties) and the methods (functions)that operate on that data into a single unit called class. By doing so it restricts direct access to some of an object’s components and prevents the accidental modification of data.