James Sharma

September 4, 2024

Foundations of Programming: Python

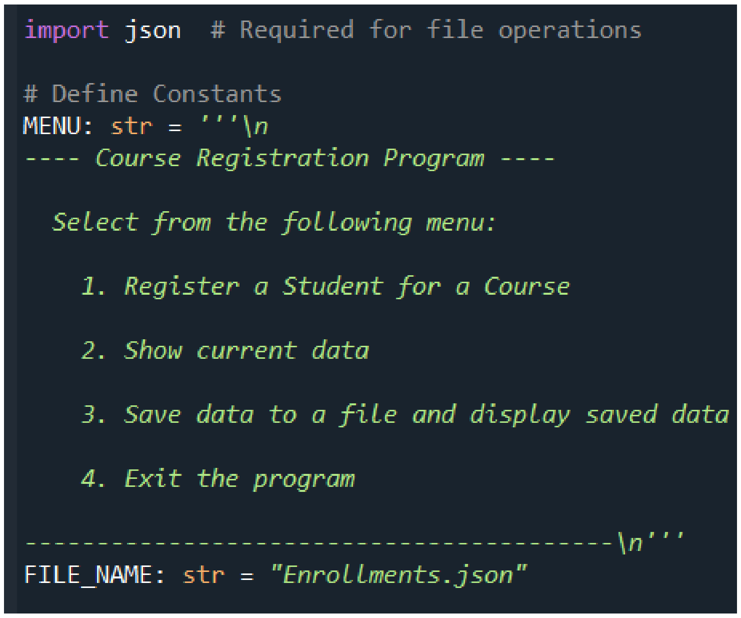
Assignment 6

Introduction

This week we examined object-oriented programming concepts and file handling. We utilized classes like IO and FileProcessor to organize the code into logical sections, each responsible for specific tasks such as user input/output and data storage. The program employs static methods to manage these tasks efficiently, ensuring code reusability and clarity. Additionally, we explored techniques like error handling with try-except blocks and data serialization with JSON to ensure reliable data management and user interaction.

JSON

JSON stands for JavaScript Object Notation. It utilizes key-value pairs. In our program, it is used to convert lists and dictionaries into text for the purpose of being stored into a file. In addition, it permits perpetuity, meaning the loaded data will carry over the next time the user opens the program. Furthermore, the text file can be converted back to Python objects. The conversion of the data into a text format is called serialization, while the transformation back to Python objects is called deserialization.



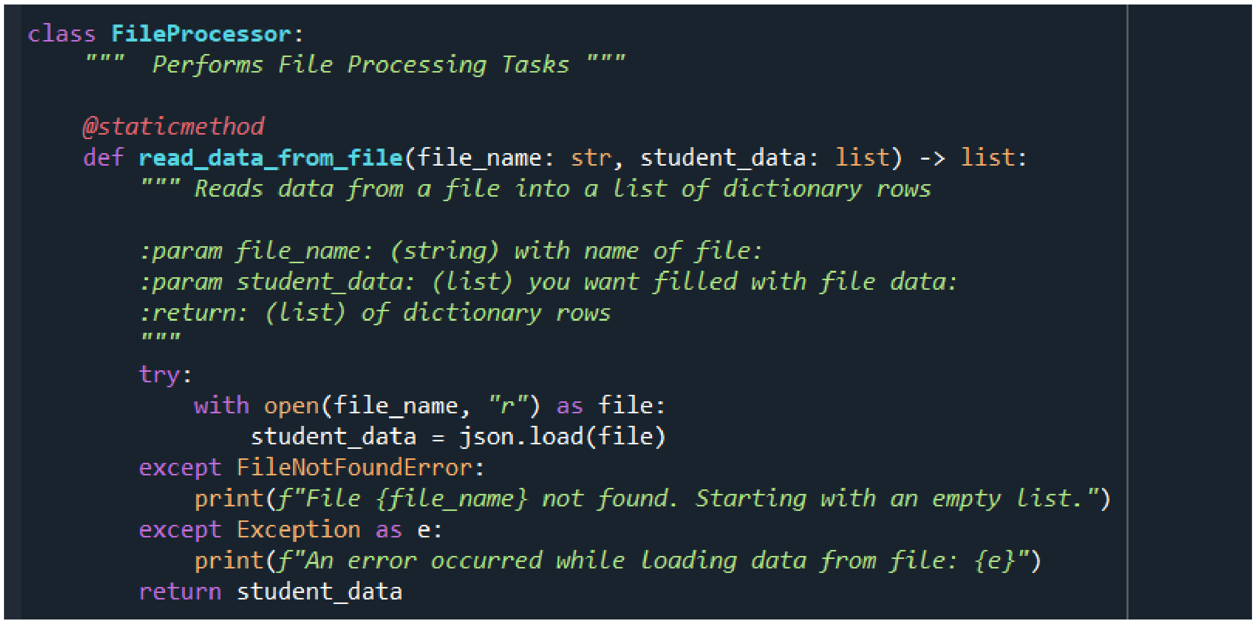
```
import json  # Required for file operations

# Define Constants
MENU: str = '''\n
---- 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 and display saved data

    4. Exit the program

-----------------------------------------\n'''
FILE_NAME: str = "Enrollments.json"
```

Figure 1: Import and Associating FILE_NAME with json

Instead of saving enrollments as a csv, we have saved it as a json file in this assignment. CSV stands for comma separated value. Hence, James, Michael, Sharma are put into different columns.

With JSON, it uses key-value pairs. For example, first name: James, middle name: Michael, last name: Sharma.

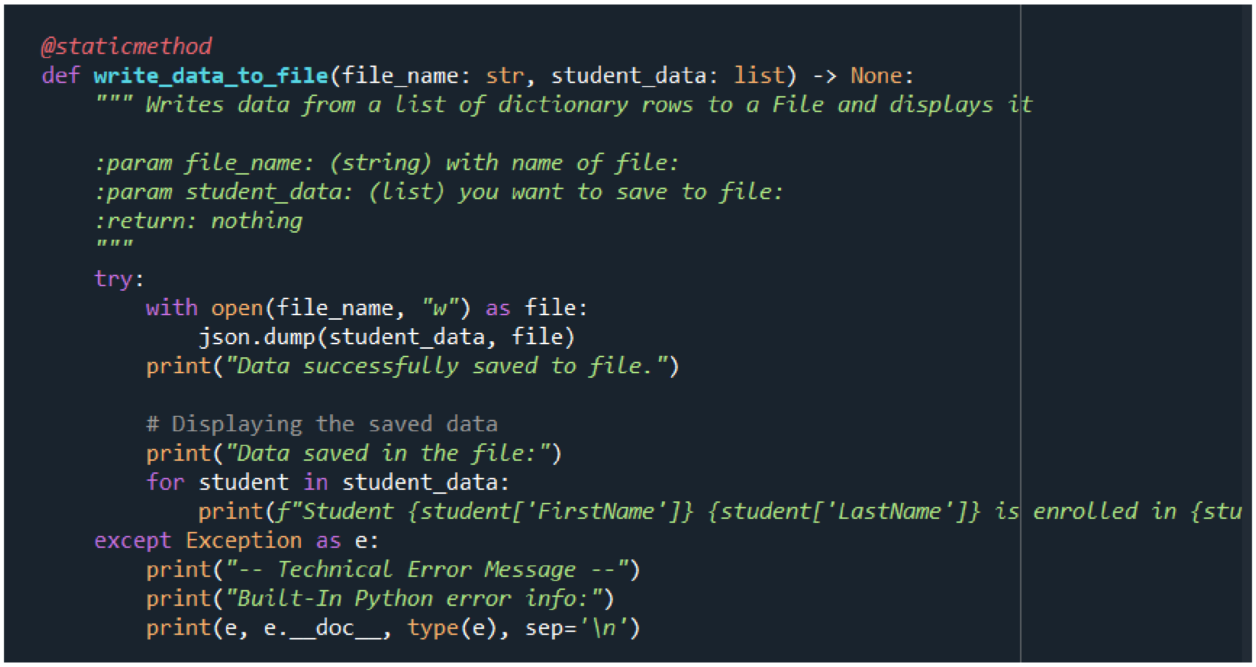
Class and Definitions



```python
class FileProcessor:
    """  Performs File Processing Tasks """

    @staticmethod
    def read_data_from_file(file_name: str, student_data: list) -> list:
        """ Reads data from a file into a list of dictionary rows

        :param file_name: (string) with name of file:
        :param student_data: (list) you want filled with file data:
        :return: (list) of dictionary rows
        """
        try:
            with open(file_name, "r") as file:
                student_data = json.load(file)
        except FileNotFoundError:
            print(f"File {file_name} not found. Starting with an empty list.")
        except Exception as e:
            print(f"An error occurred while loading data from file: {e}")
        return student_data
```

Figure 2: FileProcessor Class

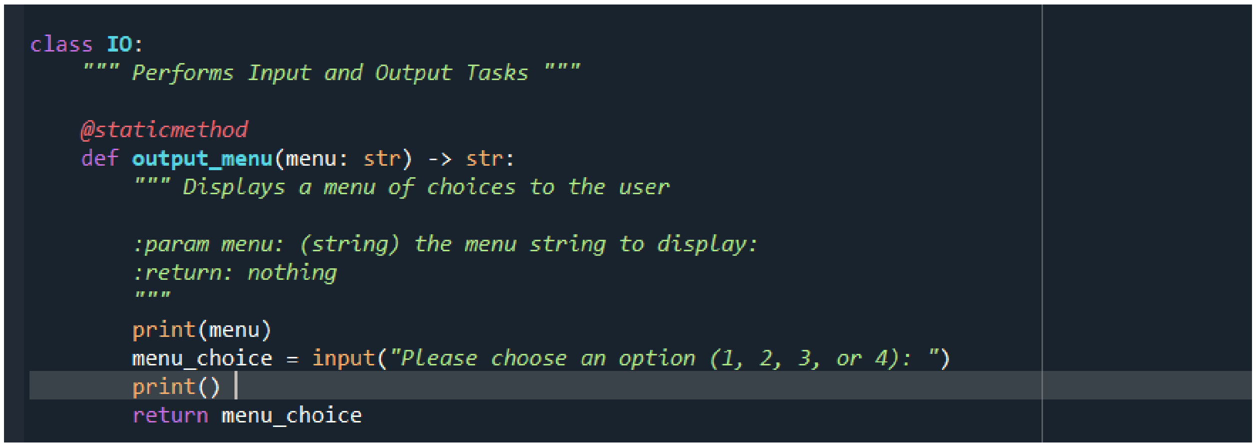
The FileProcessor class in the image contains a static method called read_data_from_file that is created to read data from the file and convert it into a list of dictionaries. To note, json is structured to read key-value pairs, which is how JSON data is structured. The method takes two parameters: file_name, which is a string representing the name of the file to be read, and student_data, a list that will be populated with the file's data. Inside the method, it uses a try block to open the file in read mode ("r") and loads its content using json.load(), which converts the JSON data in the file into a Python list of dictionaries. If the file is not found, a FileNotFoundError is caught, and a message is printed, indicating that an empty list will be used instead. If any other error occurs, it catches the generic Exception and prints an error message detailing what went wrong. Finally, the method returns the student_data list, which now contains the loaded data or remains empty if an error occurred. This method is useful for safely loading structured data from a file while handling potential errors effectively. A class is like a recipe that tells the computer how to make something, like a virtual object with certain features and actions. A definition is a set of instructions that does one specific job, like a little helper that completes a task when asked. The main difference is that a class can create many different objects with various features, while a function just performs one task whenever it is used.



```
@staticmethod
def write_data_to_file(file_name: str, student_data: list) -> None:
    """ Writes data from a list of dictionary rows to a File and displays it

    :param file_name: (string) with name of file:
    :param student_data: (list) you want to save to file:
    :return: nothing
    """
    try:
        with open(file_name, "w") as file:
            json.dump(student_data, file)
        print("Data successfully saved to file.")

        # Displaying the saved data
        print("Data saved in the file:")
        for student in student_data:
            print(f"Student {student['FirstName']} {student['LastName']} is enrolled in {stu
    except Exception as e:
        print("-- Technical Error Message --")
        print("Built-In Python error info:")
        print(e, e.__doc__, type(e), sep='\n')
```

Figure 3: definition write_data_to_file

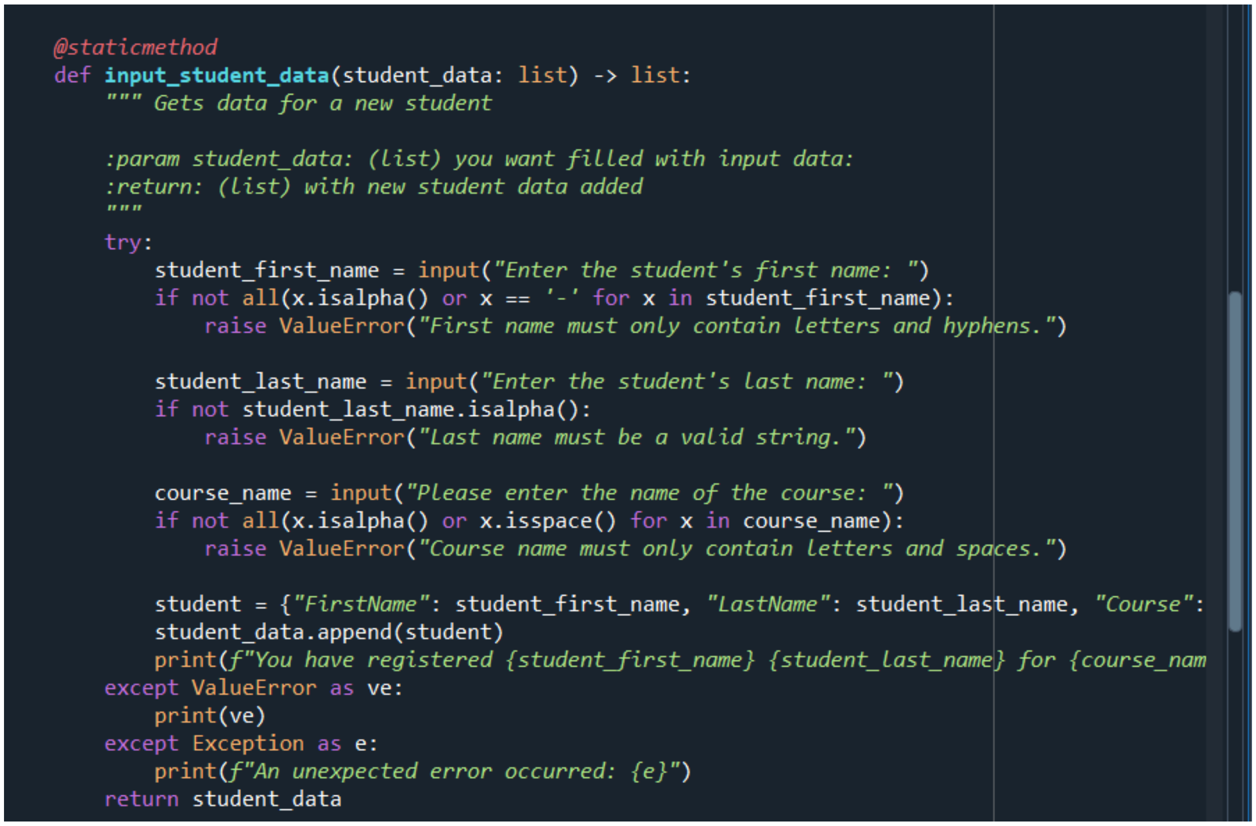
The code defines a static method named write_data_to_file within a class, which takes two parameters: file_name (a string representing the name of the file where data will be saved) and student_data (a list of dictionaries containing student information). The method's purpose is to write the provided student data into a specified file in JSON format. It opens the file in write mode ("w") using a with statement to ensure proper file handling and uses json.dump() to serialize the student_data list into JSON and save it to the file. After successfully saving the data, it prints a confirmation message and displays each student's information to the console to confirm what was saved. If any errors occur during this process, the except block catches the exceptions and prints a detailed technical error message, including the exception type and description, to help diagnose the issue. This method ensures data is saved correctly while also providing error information if something goes wrong.



```
class IO:
    """ Performs Input and Output Tasks """

    @staticmethod
    def output_menu(menu: str) -> str:
        """ Displays a menu of choices to the user

        :param menu: (string) the menu string to display:
        :return: nothing
        """
        print(menu)
        menu_choice = input("Please choose an option (1, 2, 3, or 4): ")
        print()
        return menu_choice
```

Figure 4: Displaying and Handling User Menu Choices with the IO Class

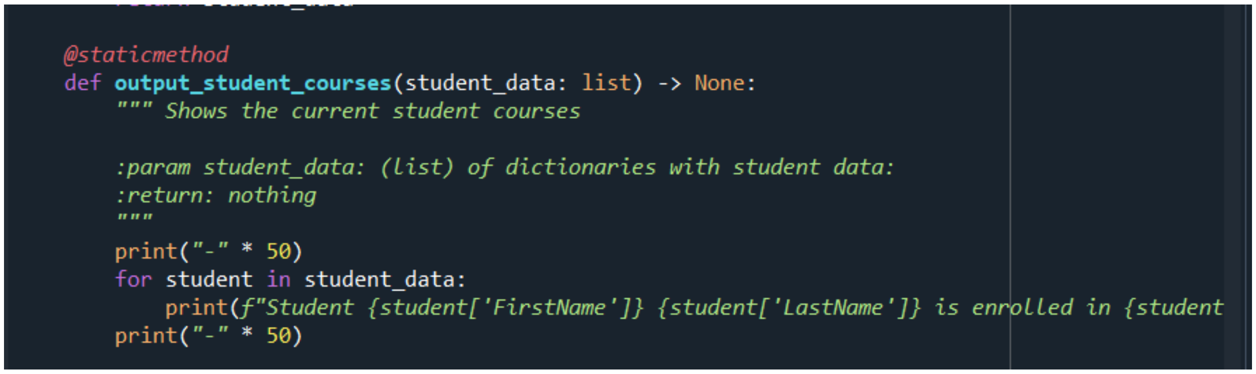
The image shows a static method called output_menu defined within the IO class, which is responsible for managing input and output tasks in the program. This method takes a single parameter, menu, which is a string representing the menu options to be displayed to the user. The method first prints the provided menu to the console, presenting the user with the available choices. It then uses the input() function to prompt the user to select an option by entering a number (1, 2, 3, or 4), which is stored in the variable menu_choice. A blank line is printed afterward to make the output more readable. Finally, the method returns the user's choice, menu_choice, allowing the program to respond to the user's selection accordingly. This method provides a simple way to interact with users, guiding them through the program's functionality and capturing their input for further processing.



```
@staticmethod
def input_student_data(student_data: list) -> list:
    """ Gets data for a new student

    :param student_data: (list) you want filled with input data:
    :return: (list) with new student data added
    """
    try:
        student_first_name = input("Enter the student's first name: ")
        if not all(x.isalpha() or x == '-' for x in student_first_name):
            raise ValueError("First name must only contain letters and hyphens.")

        student_last_name = input("Enter the student's last name: ")
        if not student_last_name.isalpha():
            raise ValueError("Last name must be a valid string.")

        course_name = input("Please enter the name of the course: ")
        if not all(x.isalpha() or x.isspace() for x in course_name):
            raise ValueError("Course name must only contain letters and spaces.")

        student = {"FirstName": student_first_name, "LastName": student_last_name, "Course":
        student_data.append(student)
        print(f"You have registered {student_first_name} {student_last_name} for {course_nam
    except ValueError as ve:
        print(ve)
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
    return student_data
```

Figure 5: Capturing and Validating Student Data with the IO Class

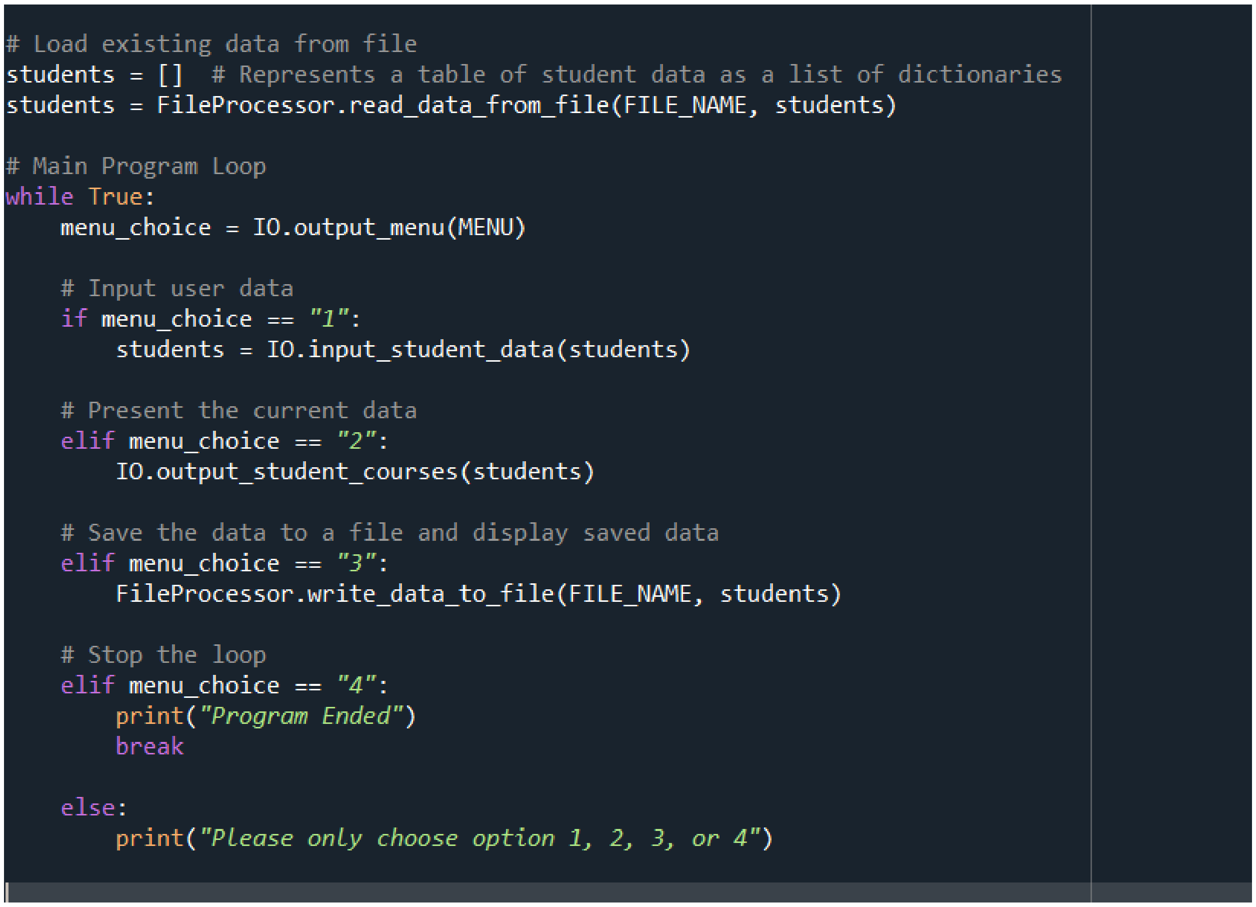
The image displays a static method named input_student_data within the IO class, which is responsible for collecting and validating information for new student registrations. The method accepts one parameter, student_data, which is a list that will be updated with new student entries. Inside the method, a try block is used to capture user input for the student's first name, last name, and course name. Each input is validated to ensure it contains only appropriate characters: the first name may contain letters and hyphens, the last name must be alphabetic, and the course name can only include letters and spaces. If any input fails validation, a ValueError is raised with a relevant message. If the input is valid, the student information is stored in a dictionary and appended to the student_data list. A confirmation message is then printed, indicating that the student has been registered. The method handles potential errors using except blocks: a ValueError provides specific feedback if input validation fails, while a general Exception catches any other unexpected errors. Finally, the method returns the updated student_data list, ensuring the new data is included. This approach ensures robust input validation and error handling when collecting new student information.



```
    @staticmethod
    def output_student_courses(student_data: list) -> None:
        """ Shows the current student courses

        :param student_data: (list) of dictionaries with student data:
        :return: nothing
        """
        print("-" * 50)
        for student in student_data:
            print(f"Student {student['FirstName']} {student['LastName']} is enrolled in {student
        print("-" * 50)
```

Figure 6: Displaying Enrolled Student Courses with the IO Class

The image shows a static method named output_student_courses within the IO class, designed to display the list of students and their enrolled courses. This method takes one parameter, student_data, which is a list of dictionaries where each dictionary contains information about a student, such as their first name, last name, and the course they are enrolled in. The method starts by printing a line of dashes ("-" * 50) to visually separate the output, making it more readable. It then iterates over each student dictionary in the student_data list using a for loop. For each student, it prints a formatted string that shows the student's full name and the course they are enrolled in, providing a clear and organized display of all registered students and their courses. After the loop completes, it prints another line of dashes to close off the section. This method does not return any value (None); instead, it directly outputs the formatted list of students and courses to the console, making it a simple and effective way to present the current enrollment data.



```python
# Load existing data from file
students = []  # Represents a table of student data as a list of dictionaries
students = FileProcessor.read_data_from_file(FILE_NAME, students)

# Main Program Loop
while True:
    menu_choice = IO.output_menu(MENU)

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

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

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

    # Stop the loop
    elif menu_choice == "4":
        print("Program Ended")
        break

    else:
        print("Please only choose option 1, 2, 3, or 4")
```

Figure 7: Main Program Loop for Managing Student Registrations

The image shows the main program loop for managing student registrations, which continuously prompts the user for input until the user decides to exit. Initially, the program starts by loading any existing student data from a file using the FileProcessor.read_data_from_file method, which populates the students list with data. This list represents a table of student data as a list of dictionaries. The while True loop runs indefinitely until explicitly broken. Within the loop, the IO.output_menu method displays a menu and captures the user's choice, storing it in menu_choice. Depending on the user's choice, the program performs different actions: if the user selects "1," it collects new student data via IO.input_student_data; if "2" is selected, it displays the current student enrollments using IO.output_student_courses; and if "3" is chosen, it saves the student data to a file and displays the saved data with FileProcessor.write_data_to_file. If the user enters "4," the program prints "Program Ended" and breaks the loop, effectively stopping the program. Any invalid input triggers an error message prompting the user to choose a valid option. This structured loop ensures a smooth user experience for managing student data.

Conclusion

By implementing these concepts, we learned how to create an effective application that manages student data through a user-friendly interface. The structured approach using classes and static methods made the code organized and easier to maintain, while JSON serialization provided a simple way to store and retrieve complex data. This exercise highlighted the importance of combining object-oriented design, error handling, and data management techniques to develop interactive software solutions. Overall, the project demonstrated how these programming principles can be applied to solve real-world problems efficiently.