**Case Study 3: Design and Implementation of an Object-Oriented**

**E-Learning Platform**

**Case Study Submitted to Professor Joseph C. Lorilla**

**College of Engineering and Information Technology**

**University of Southern Mindanao**

**OBJECT ORIENTED PROGRAMMING**

**BACHELOR OF SCIENCE IN COMPUTER SCIENCE**

**Rey Intes**

**Anecito Clatero Jr.**

**Charlotte Carmona**

**A yellow and green logo

Description automatically generated**

**December 2024**

1. **DESCRIPTION**

This case study outlines the development of a comprehensive **E-Learning Platform** designed to manage various aspects of an integrated set of interactive online services that provide **PlatformAdmin**, **Instructors**, **Students,** and others involved in education with information, tools, and resources to support and enhance education delivery and management. At its core, the system features a structured class hierarchy with well-defined responsibilities for each component, utilizing object-oriented principles such as **inheritance, encapsulation, and polymorphism**.

The base of the system is the **Person** class, which serves as a foundation for managing general functionality. This class branches into two specialized subclasses: **student** and **instructor.** Students are the primary users who access course materials, participate in lessons, complete assignments, and engage with other learning resources. Instructors create and organize course content, lessons, quizzes, and assignments.

The **PlatformAdmin** class manages user accounts, including students, instructors, and other staff members. They can create, modify, or deactivate user profiles. The **Course** classes are created by instructors or the platform administrators, and they consist of structured content such as modules, units, or sections, which helps students navigate through the content easily. The **Enrollment** class is so helpful because students can enroll in courses that are available to them based on prerequisites, availability, and their interests. Enrollment can be automated or managed manually, and the platform and instructors can monitor who is enrolled in which courses and track the status (e.g., active, completed).

**Attendance** class is a mechanism for tracking student presence or absence in a subject within a course. Students can mark their attendance, and instructors can view attendance records for their courses. The class itself manages the interaction between students and the data storage for attendance, providing functionality for both students to mark their attendance and for instructors to view it. For the **Assignment** class, the instructors create assignments, quizzes, or projects within the platform. These can be formative (ongoing assessments) or summative (final exams). Assignments are designed to assess students’ understanding of the course material.

While for the **Grade** class, the instructors assess student performance through assignments, quizzes, exams, and projects. Grades are recorded and stored within the platform; also, the students can view their own grades and track their performance, while instructors and administrators can monitor overall class performance and trends. For **Schedule** class, the platform administrator and instructors set up schedules for live classes, webinars, or exams, which students need to follow. The platform may display the timing of lectures, assignments, and deadlines.

Additionally, the **DataManager** is responsible for organizing and storing all user-related data, course content, assessments, grades, and other relevant information. It ensures that data is easily accessible and securely stored. The **Login** class ensures that only authorized users (students, instructors, admins) can access the platform. Users must provide credentials (username and password) to log in.

The **Info** class is about each course, such as its title, description, syllabus, schedule, prerequisites, and instructors, which is displayed in the course catalogue or course landing pages. This helps students decide which courses to enrol in.

To safeguard sensitive information, **encapsulation** is implemented through access modifiers (public, private, and protected), controlling access to attributes and methods as needed. The system also employs **static methods** for order and payment tracking, enabling shared functionalities across multiple instances without requiring instantiation.

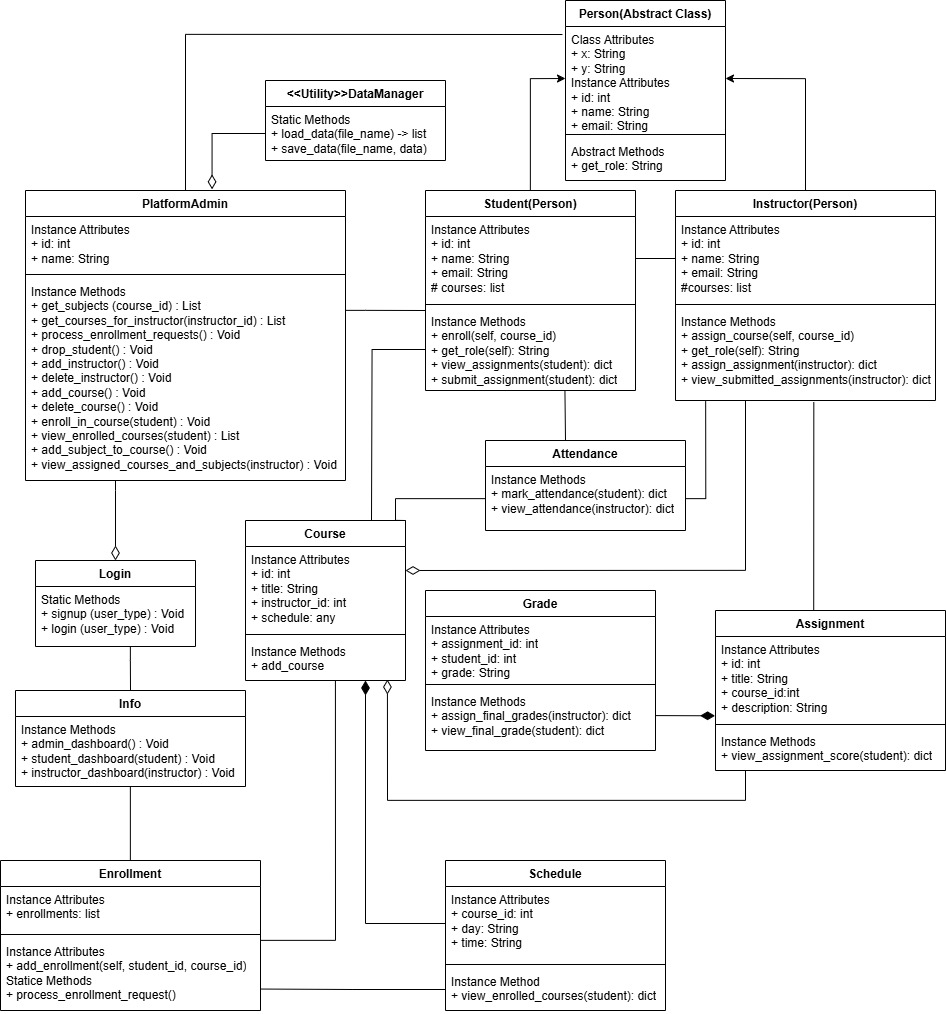
**System Requirements**

The E-Learning Platform includes key requirements, such as:

* **User Management**: Focuses on user accounts and permissions.
* **Course Manager**: Handles course creation, organization, and updates.
* **Enrollment Manager**: Oversees the process of course registration and student access.
* **Instructor Manager**: Supports instructors, ensuring they can effectively teach and manage their courses.
* **Grade Manager**: Manages grading systems, tracks student performance, and generates reports.

This **e-learning platform** case study highlights how **object-oriented programming** principles—such as **scalability**, **security**, **reusability**, **modularity**, and **customizability**—are applied to ensure a **robust, secure**, and **scalable learning environment** that can grow with the needs of users while offering an efficient and adaptable structure for development and maintenance.

1. **CLASS DIAGRAM:**



1. **CLASS DIAGRAM EXPLANATION:**

**Person Class (Abstract Base Class)**

* This class is typically an abstract class, meaning it cannot be instantiated on its own but is used as a base for other classes. It holds common attributes and methods that can be shared across different types of people (like students, instructors, etc.).

**PlatformAdmin Class**

* This class represents an administrator who has access to manage the entire platform. They are responsible for overseeing all operations, including managing users (students and instructors), courses, schedules, and possibly approving content or handling reports.

**Student Class**

* This class extends the Person class and adds student-specific functionality.

Attributes could include student\_id, enrollment\_status, and a list of courses they are enrolled in.

**Instructor Class**

* This class also extends the Person class and adds instructor-specific functionality.

**Assignment Class**

* This class represents a task or project that students need to complete as part of a course.

**Enrollment Class**

* This class keeps track of which students are enrolled in which courses.

**Attendance Class**

* This class itself manages the interaction between students and the data storage for attendance, providing functionality for both students to mark their attendance and for instructors to view it.

**Grade Class**

* This class holds the grading information for assignments or entire courses.

**Schedule Class**

* This class represents the schedule for courses, instructors, and students.

**Login Class**

* This class handles the login functionality for the platform.

**DataManager Class**

* This class is responsible for managing all data-related tasks, such as storing, retrieving, and processing data from a JSON file.

**Course Class**

* This class represents a specific course that is offered on the platform.

**Info Class**

* This class could be used to store and display informational content about the platform, such as FAQs, system guidelines, or course descriptions.

**Person (ABC)** <- **Student** <- **Instructor**

The abstract base class for both Student and Instructor. It defines a common structure for them. Student and Instructor inherit from Person.

**Student** -> **Course** -> **Enrollment**

Has a relationship with Course through the enroll method, which connects students to courses. Can submit assignments via the submit\_assignment method, which connects with the Assignment class.

**Instructor** <>-- **Course**

An instructor is related to a Course, as an instructor is assigned to teach a course. However, a Course can exist without an instructor (e.g., before an instructor is assigned).

**Instuctor** -> **Assignment** -> **Grade**

Can assign assignments via the assign\_assignment method, linking to the Assignment class. Views submitted assignments via the view\_submitted\_assignments method, which connects to the Assignment and Grade classes.

**Course** -> **Instructor**

Links with Instructor through instructor\_id, defining which instructor teaches the course.

**Course** -> **Schedule**

Has multiple Schedule entries, representing the timetable.

**Course** <>-- **Assignment**

A Course can have many Assignments, but assignments exist independently of a specific course. Even if a course does not have assignments assigned at a given time, the assignment can be linked to a different course.

**Enrollment** -> **Course** -> **Student**

Contains a list of student-course relationships, helping to associate a student with the courses they are enrolled in.

**Assignment** -> **Course**

Contains assignments for specific courses

**Assignment** ◆--**Grade**

An Assignment has a Grade assigned to it. If the Assignment is removed (for example, if the assignment is deleted or changed), the Grade would no longer be valid.

**Grade** <- **Assignment** <- **Student**

Represents the grade for a student's assignment submission.

**Schedule** --◆ **Course**

A Course has multiple Schedules, and these schedules are directly tied to the course. If the course is deleted, the schedules associated with that course would also typically be deleted.

**DataManager** -> **Instructor** -> **Student** -> **Course** -> **Assignment**

Provides utility functions to load and save data, affecting multiple classes like Course, Assignment, Instructor, and Student.

**PlatformAdmin** <>-- **DataManager**

PlatformAdmin interacts with DataManager to load and save data for various entities (students, instructors, courses). DataManager is not dependent on the PlatformAdmin for its existence, meaning DataManager can function independently.

**PlatformAdmin** -> **Student** -> **Instructor** -> **Courses**

It provides methods that handle the management of students, instructors, courses, and subjects.

**PlatformAdmin** -> **Login** -> **Info** -> **Instructor** -> **Student** -> **Course**

It communicates with other components like Login, Info, Instructor, Student, and Course.

**Login** <>-- **PlatformAdmin**

A Login process can lead to a PlatformAdmin's dashboard, but a Login exists independently of a PlatformAdmin. The Login class facilitates access to the platform but is not inherently bound to a particular role (it could be a student or instructor as well).

**Login** -> **Info**

It communicates with Info to take users to their respective dashboards based on their type.

**Info** -> **PlatformAdmin** -> **Student** -> **Instructor**

Handles the navigation to the respective dashboards (platformadmin\_dashboard, student\_dashboard, and instructor\_dashboard) after a successful login. The Info class directs the flow of interactions based on the user role.

**Student** -> **Assignment**

Represents student-specific features such as viewing assignments, submitting assignments, and viewing enrollment status.

**Student** -> **PlatformAdmin**

The student interacts with the PlatformAdmin for enrollment and course-related functionalities.

**Instructor** -> **Assignment**

Manages assignment-related actions, including assigning and viewing submitted assignments.

**Instructor** -> **PlatformAdmin**

The instructor also interacts with PlatformAdmin to view and manage the courses and subjects they are assigned to.

**Course** -> **Instructor**

Represents course-related data and structure such as course ID, title, and the associated instructor and subjects.

**Student** -> **Attendance**:

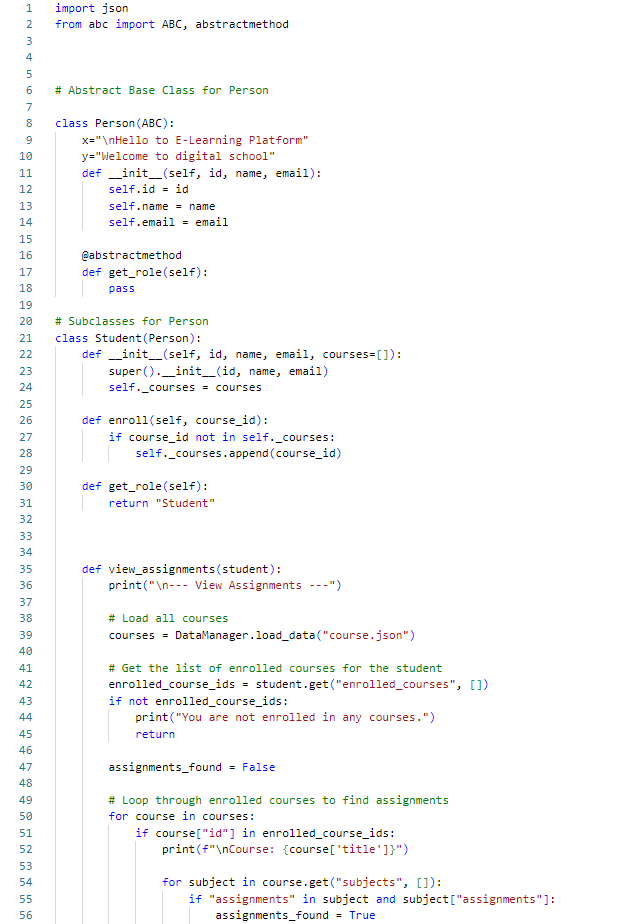
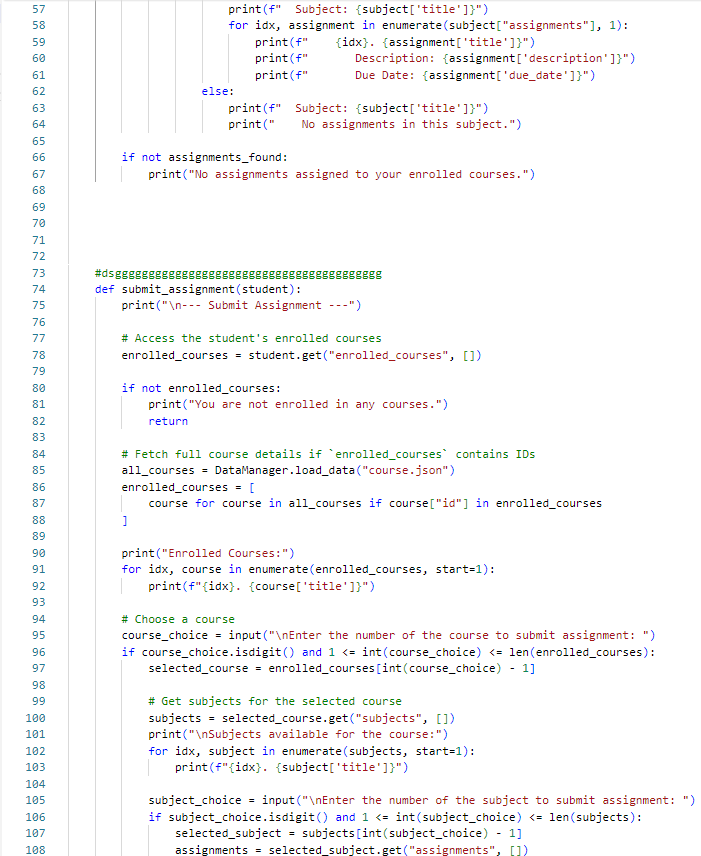
It is where a student interacts with the Attendance class to mark their attendance.

**Instructor** -> **Attendance**

Where an instructor views attendance records for their assigned subjects.

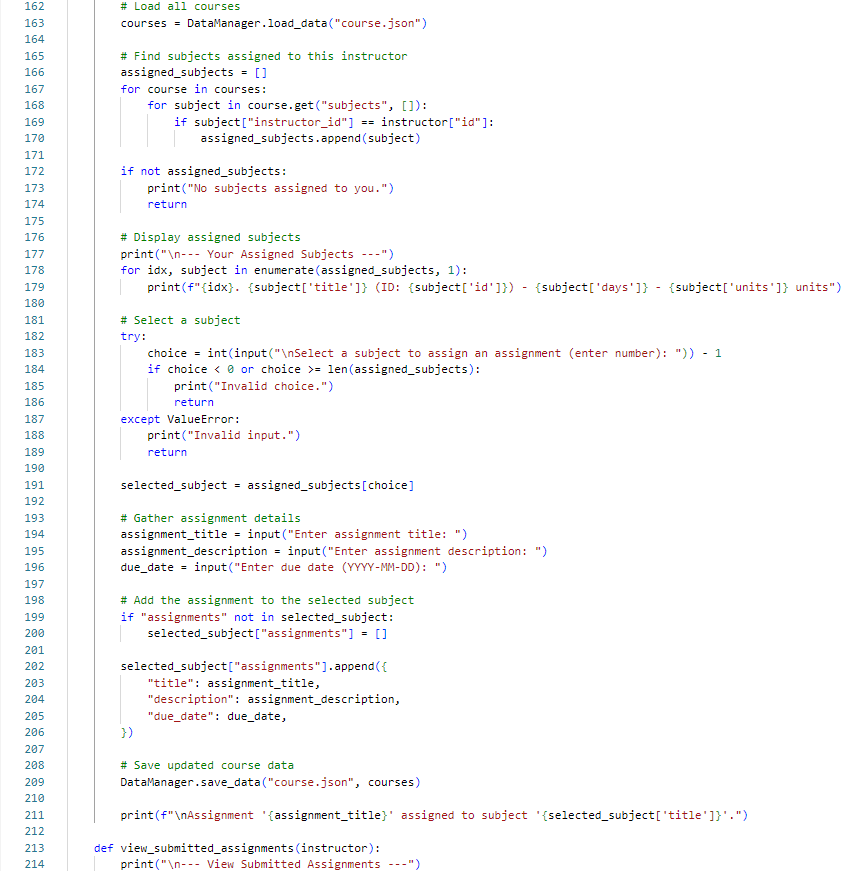
**Course** -> **Attendance** (via Subject)

It’s where attendance is tracked for courses and subjects, but the Course class isn't directly responsible for handling attendance. The Attendance class contains the references to both the course\_id and subject\_id to link the attendance to a specific course and subject.

1. **CODE SNIPPETS & EXPLANATION:**

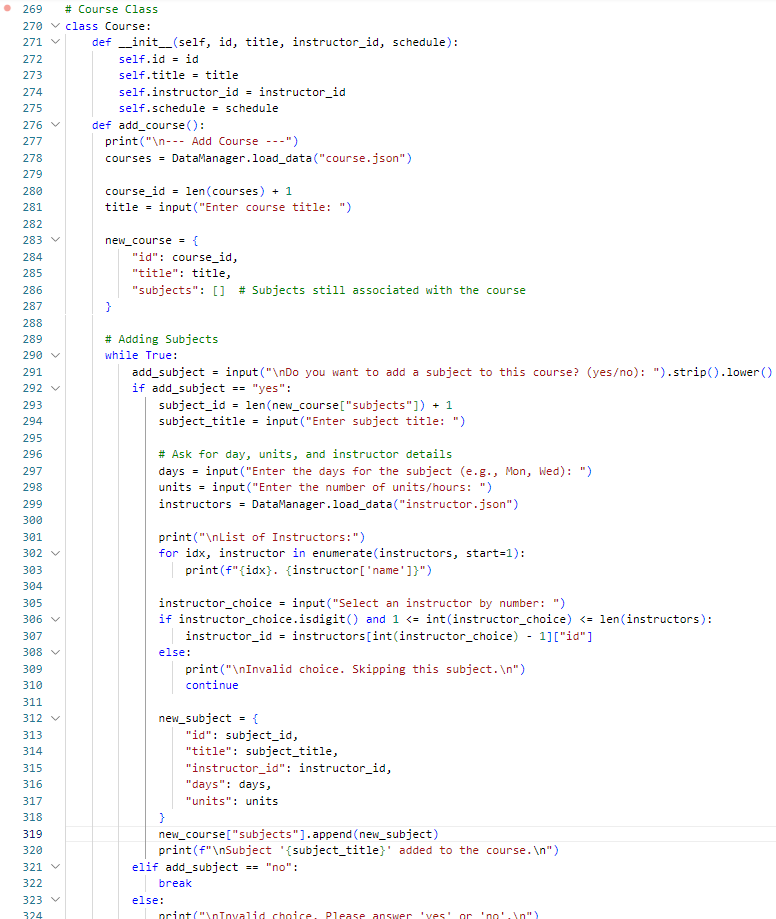
This part of the code prints out the assignments for each subject in a course. It first prints the title of the subject **(subject['title']).** Then, for each assignment in that subject **(subject["assignments"]),** it prints: The assignment number (starting from 1, due to **enumerate**). The title, description, and due date of each assignment. This function handles the process of submitting an assignment. It starts by printing the header **--- Submit Assignment ---.** It then tries to get the list of enrolled courses from the **student** using **student.get("enrolled\_courses", []).** The **get** method will return an empty list if no enrolled courses are found. The code loads all course data from a file called **course.json** using **DataManager.load\_data("course.json").** It then filters out the courses that the student is enrolled in by comparing the course IDs **(course["id"])** with the **enrolled\_courses** list. The function allows the student to select a course, then a subject within that course. For each subject, it shows the list of assignments and asks the student to choose one to submit. The **enrolled\_courses**, **subjects**, and **assignments** are fetched dynamically from the course data, which is loaded from **course.json.**

This code defines a simple e-learning platform model using Python's Object-Oriented Programming **(OOP)** concepts. This is an abstract base class, meaning that it cannot be instantiated directly. Other classes (like **Student**) will inherit from this class. Then the attributes **x** and **y** are class-level variables (strings) that hold messages, presumably for display. The ***\_\_init\_\_*** method initializes the **id**, **name**, and **email** attributes when a **Person** object is created. Then the methods **get\_role** is an abstract method, which means any subclass (like **Student**) must implement this method. It is meant to return the role of the person (like "Student" or "Teacher"). This is a subclass of **Person**, meaning a **Student** is a type of **Person**. It inherits attributes and methods from **Person**. ***\_\_init\_\_*** method takes the same **id**, **name**, and **email** attributes as the Person class, but also a list of **courses** that the student is enrolled in. The **enroll** method adds a **course\_id** to the **Student's** list of enrolled courses, but only if the **course\_id** is not already in the list. The **get\_role** method overrides the abstract **get\_role** method in the **Person** class and returns the string **"Student"**, indicating that this person is a student. **Student** class allows enrolling in courses and has a method to view assignments. The **view\_assignments** method attempts to display assignments for the courses a student is enrolled in, though it contains a couple of issues that need to be addressed for the code to function properly.

****

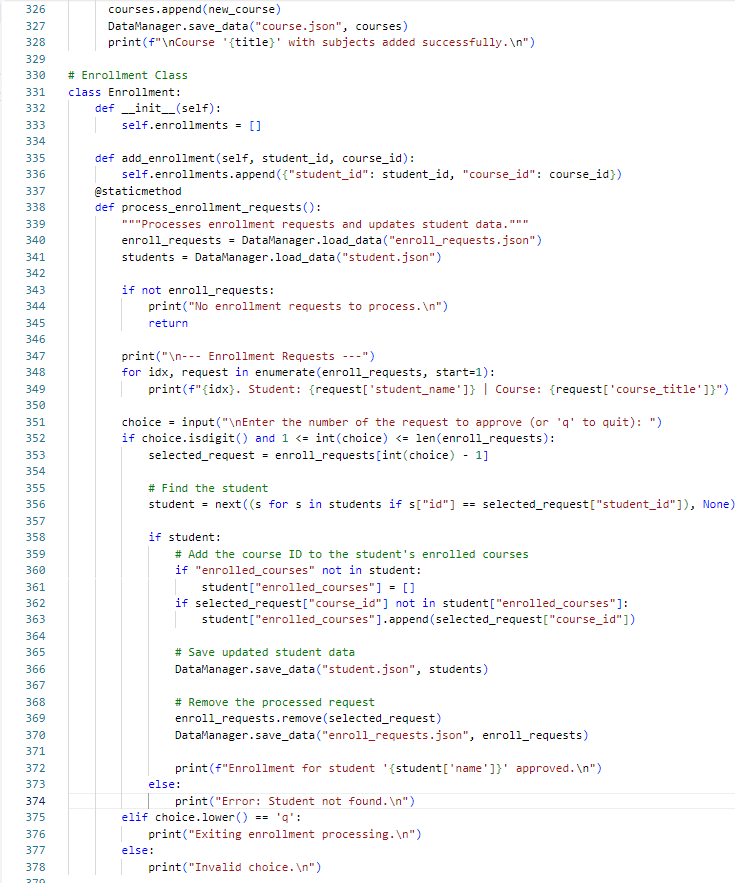
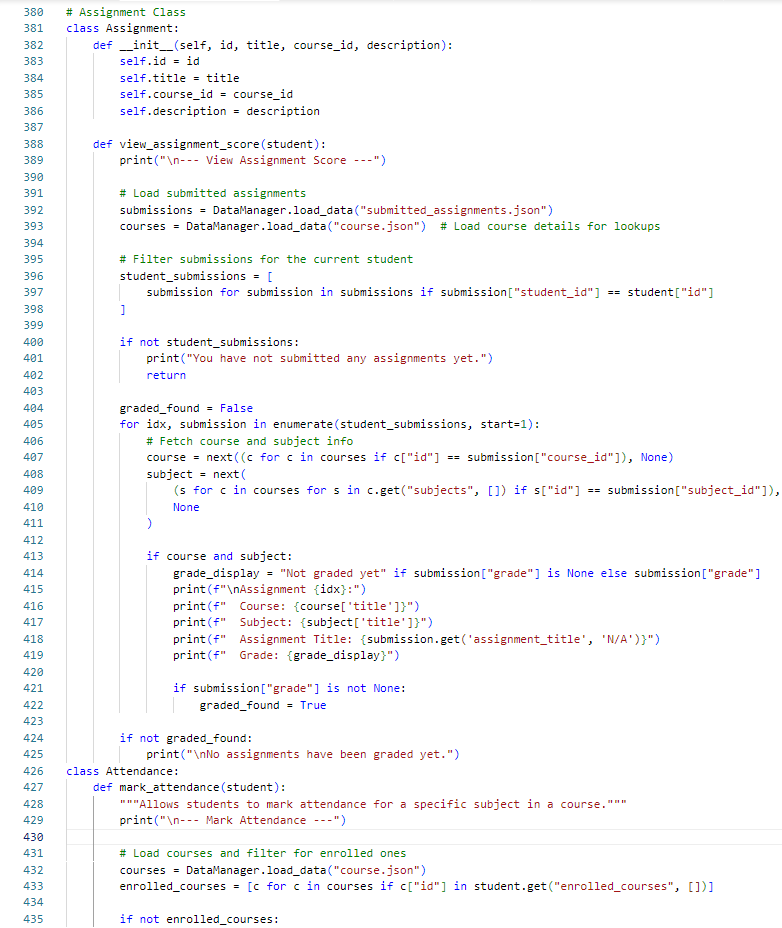
This line loads the course data from a file called **course.json** using the **DataManager.load\_data()** method. It assumes that **course.json** contains details about all the courses and their associated subjects, including instructor information. This part of the code loops through all the courses and their subjects to find the ones assigned to the current instructor. The code checks each course’s **subjects** list and compares the **instructor\_id** of each subject to the **instructor["id"]** (the ID of the instructor currently using the system). If the instructor is assigned to a subject, that subject is added to the **assigned\_subjects** list. After the loop, it checks if the **assigned\_subjects** list is empty. If no subjects are assigned to the instructor, it prints a message ("No subjects assigned to you.") and exits the function. If the instructor has subjects assigned, the code prints out the details of each assigned subject. The instructor is asked to select a subject by entering a number corresponding to one of the listed subjects. This line selects the subject that corresponds to the user's choice. The instructor is asked to provide the details for the new assignment they want to assign. The code first checks if the selected subject already has an **assignments** key. If not, it initializes it as an empty list **(selected\_subject["assignments"] = []).** After the assignment has been added, the updated course data (which now includes the new assignment) is saved back to the **course.json** file using **DataManager.save\_data().** This is the beginning of the **view\_submitted\_assignments** method, which seems to be designed for the instructor to view the assignments that students have submitted.

If there are **assignments** available for the selected subject, the code lists them. It checks if the assignments list is not empty. Then, it enumerates through the **assignments** list, printing the assignment index, title, and due date for each assignment. The input is validated to ensure it is a digit and within the range of available assignments. If valid, it selects the assignment from the list **(selected\_assignment).** This part handles saving the assignment submission. It loads existing submission data from **submitted\_assignments.json** using **DataManager.load\_data.** It creates a submission\_data dictionary containing **student's ID**, **name**, **course ID**, **subject ID** for the assignment. It then saves the updated list of submissions back to the **submitted\_ assignments.json** file using **DataManager.save\_data.** The instructor class inherits from the Person class and represents an instructor. The ***\_\_init\_\_*** method takes **id**, **name**, **emai**l, and a list of courses (defaulting to an empty list) as arguments. The super().***\_\_init\_\_(*id, name, email)** line calls the **parent** **class** Person's constructor to initialize the **id**, **name**, and **email** attributes. This method allows an instructor to assign a course to themselves by adding a **course\_id** to their list of courses. This method overrides the abstract **get\_role** method from the **Person** class to return **"Instructor",** indicating the role of this person is an instructor. This is a method intended for the instructor to assign assignments. It prints a header message indicating the start of the assignment assignment process.

****

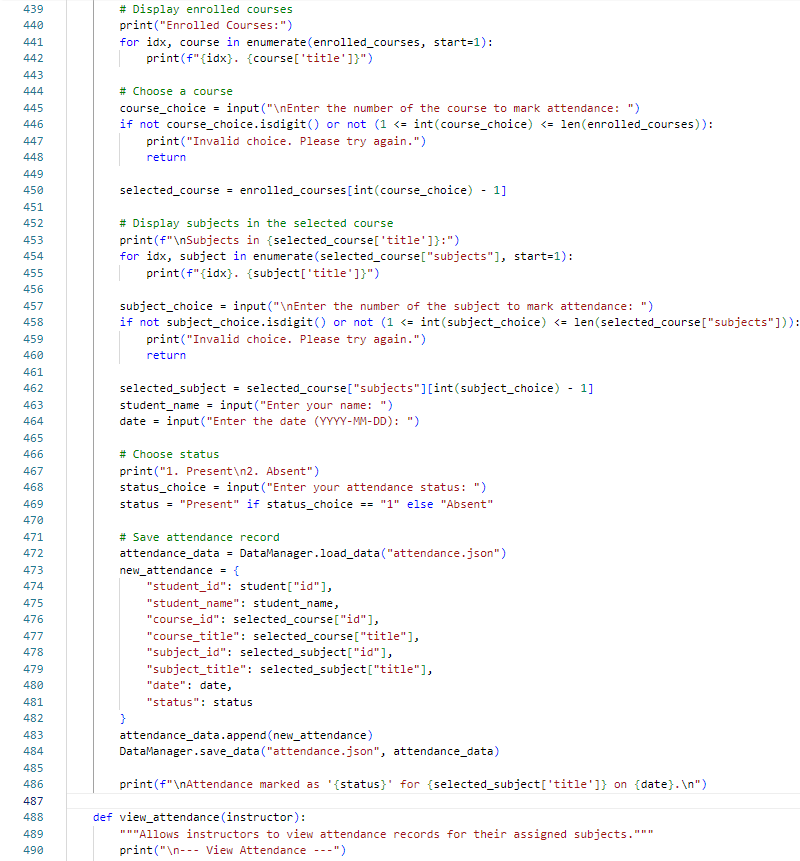
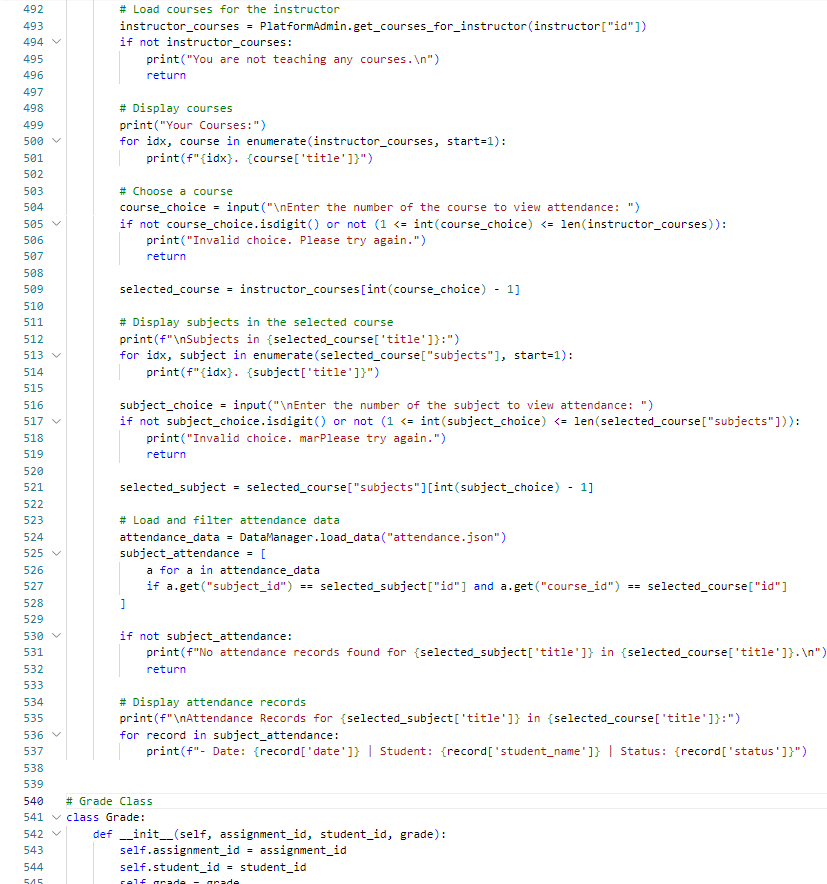
This is the constructor (***\_\_init\_\_*** method) for the Course class. When you create a new instance of this class, you provide the following attributes: **id**, **title**, **instructor\_id** and **schedule**. This method is used to add a new course to the system.It starts by loading existing course data from the file **course.json** using **DataManager.load\_data().** A new course ID is generated by taking the current number of courses and adding 1. This assumes that each course in the **courses** list has a unique ID and new courses will incrementally receive new IDs. The user is prompted to enter a title for the new course. This title will be used for the new course’s **title** field. A new course is created as a dictionary with the following fields: **id**, **title** and **subjects**. This starts a loop where the user can keep adding subjects to the course. The program asks the user whether they want to add a subject to the course. The user is asked to input: The **days** when the subject will be taught (e.g., Monday, Wednesday). The **units** for the subject (e.g., 3 hours or 4 units). The system loads the list of instructors from **instructor.json**. A new subject is created as a dictionary with the following fields: **id**, **title**, **instructor\_id**, **days**, and **units**. This new subject is then added to the subjects list of the **new\_course** object. If the user responds "**no**" to adding more subjects, the loop ends, and no further subjects are added to the course.

This line JSON file called **submitted\_assignments.json** using the **DataManager.load\_data()** method. If there are no submissions (i.e., the file is empty or doesn't contain data), it prints a message saying "No assignments submitted yet." and exits the function. First, it fetches the list of courses assigned to the instructor using **PlatformAdmin.get\_courses\_for\_instructor (instructor["id"]).** Then, it creates a list of subject IDs **(instructor\_subject\_ids)** for all the subjects taught by this instructor. It checks each course's **subjects** to see if the **instructor\_id** matches the current instructor’s ID. If so, it adds the subject's ID to the list. It compares the **subject\_id** of each submission to the list of subject IDs the instructor is teaching (**instructor\_subject\_ids**). If there's a match, the submission is included in the **filtered\_submissions** list. If there are no submissions for the instructor's subjects (i.e., **filtered\_submissions** is empty), it prints a message saying "No assignments submitted for your subjects." and exits the function. This section loops through all the filtered submissions (**filtered\_submissions**) and displays the: **Student Name, Assignment Title, Answer** and **Grade**. The instructor is prompted to select a submission to grade by entering the number corresponding to the submission. After the instructor provides a grade, the code looks through the list of all submissions (**submissions**) to find the specific submission to update. After the grade has been assigned, the updated list of submissions is saved back to the **submitted\_assignments.json** file using **DataManager.save\_data().** Finally, a success message is printed to confirm that the grade was successfully assigned to the selected submission.

****

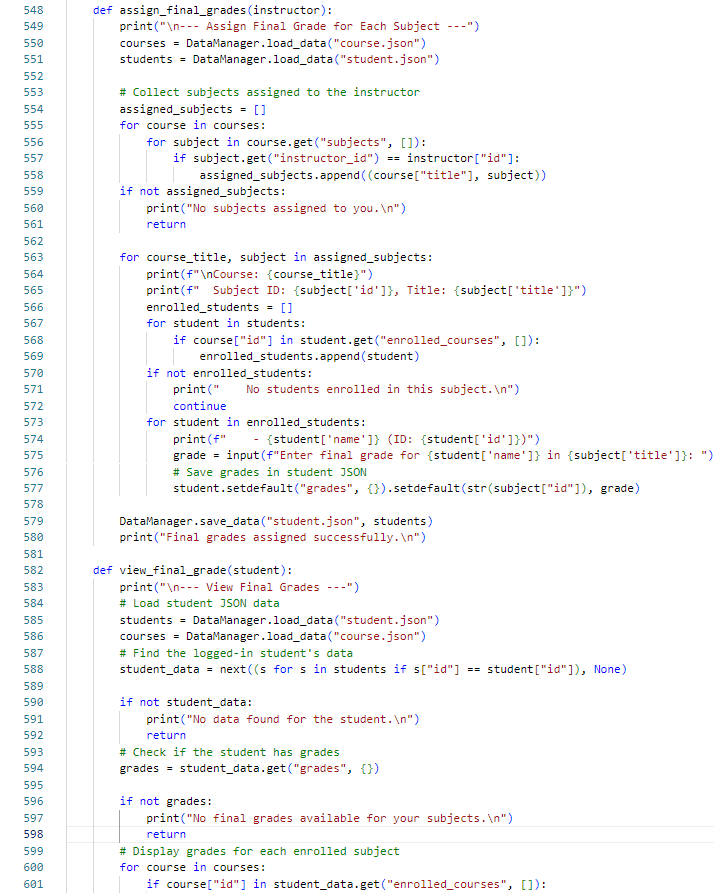
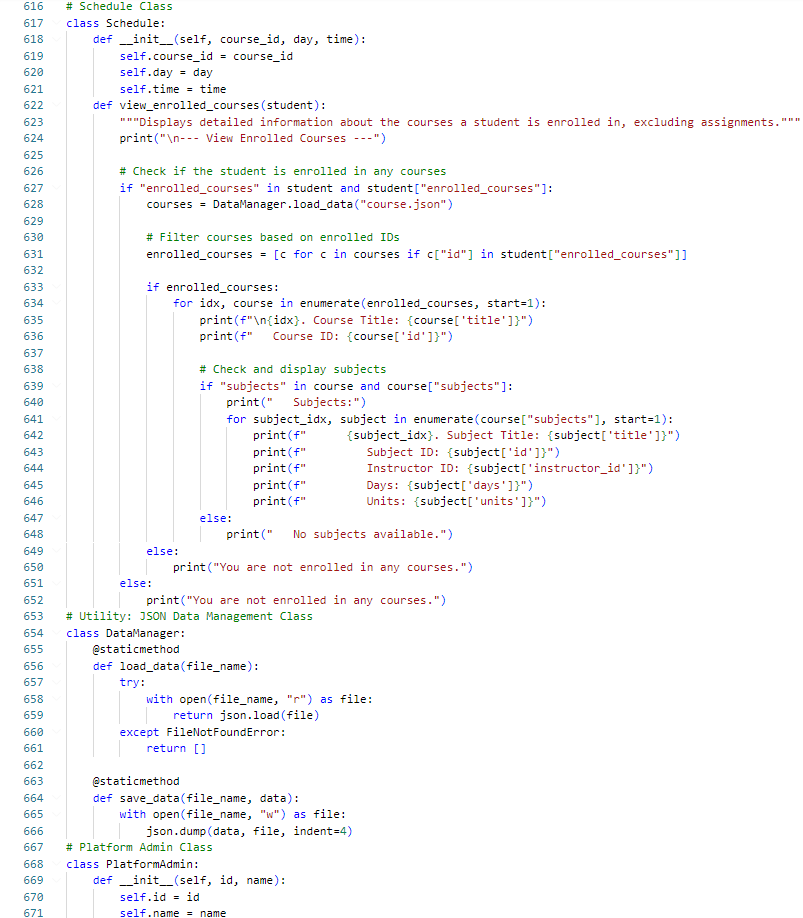
This Assignment class handles operations related to assignments, such as viewing assignment scores for students. The constructor initializes an Assignment object with the following properties: **id**, **title**, **course\_id** , and **description**. This method allows a student to view their assignment scores. The method loads two data sets: **submissions**: Contains all assignments that have been submitted by students. **courses**: Contains course details for lookup (i.e., to fetch course and subject information). Filters the list of submitted assignments to get only the ones that belong to the current student (based on **student\_id**). If the student has not submitted any assignments, the method prints a message and returns early. The method iterates through the student's submissions and attempts to find corresponding course and subject details by matching **course\_id** and **subject\_id** from the submission to the course data. If course and subject information is found, it displays: **course**, **subject titles**, **assignment's title** and the **grade** for the assignment. If the assignment has not been graded, it shows "**Not graded yet**." . If at least one assignment has been graded, **graded\_found** is set to **True**. If no graded assignments were found, a message is printed indicating that no assignments have been graded yet. This **Attendance** class handles marking attendance for students. This method allows a student to mark their attendance for a subject in a course. The method loads the course data and filters it to include only the courses that the student is enrolled in. This is done by checking if the course's **id** exists in the student's **enrolled\_courses.**

After creating the new course with its subjects, this code adds the **new\_course** to the courses list, which holds all the courses in the system. **DataManager.save\_data("course.json", courses)** saves the updated list of courses (including the new course) to the **course.json** file, so the course is persisted in the system. This Enrollment class manages the enrollment of students in courses. It stores a list of enrollments and has methods to handle them. The constructor initializes the **enrollments** list, which will hold all the enrollment records. Each record will typically contain a student ID and a course ID. This method allows a student to be enrolled in a course by appending a dictionary containing the student’s ID and the course’s ID to the **enrollments** list. It loads two sets of data: **enroll\_request** and **students**. If there are no enrollment requests, the method prints a message and exits early. It lists all the pending enrollment requests, showing the student’s name and the course title they’re requesting to enroll in. It prompts the admin to enter the number of the enrollment request they wish to approve or to type 'q' to quit the process. If the user enters a valid number corresponding to an enrollment request, the method retrieves the selected request. The method searches for the student in the **students** list by matching the student’s ID from the request. If the user types 'q', the method exits the enrollment processing loop and stops further action. If the user enters an invalid choice (anything other than a valid number or 'q'), the method prints an error message.

****

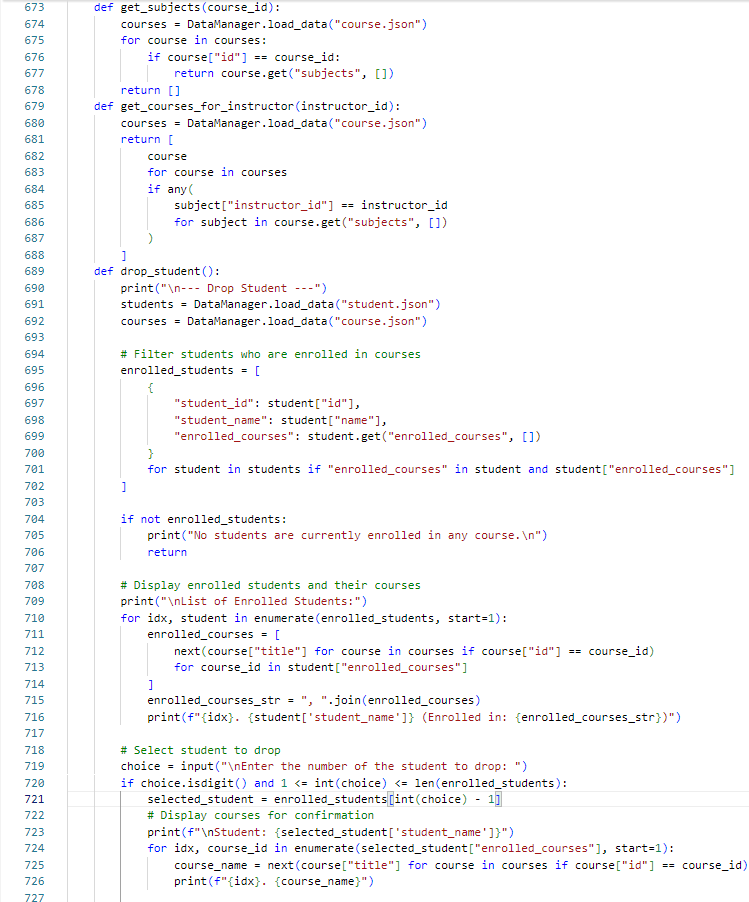
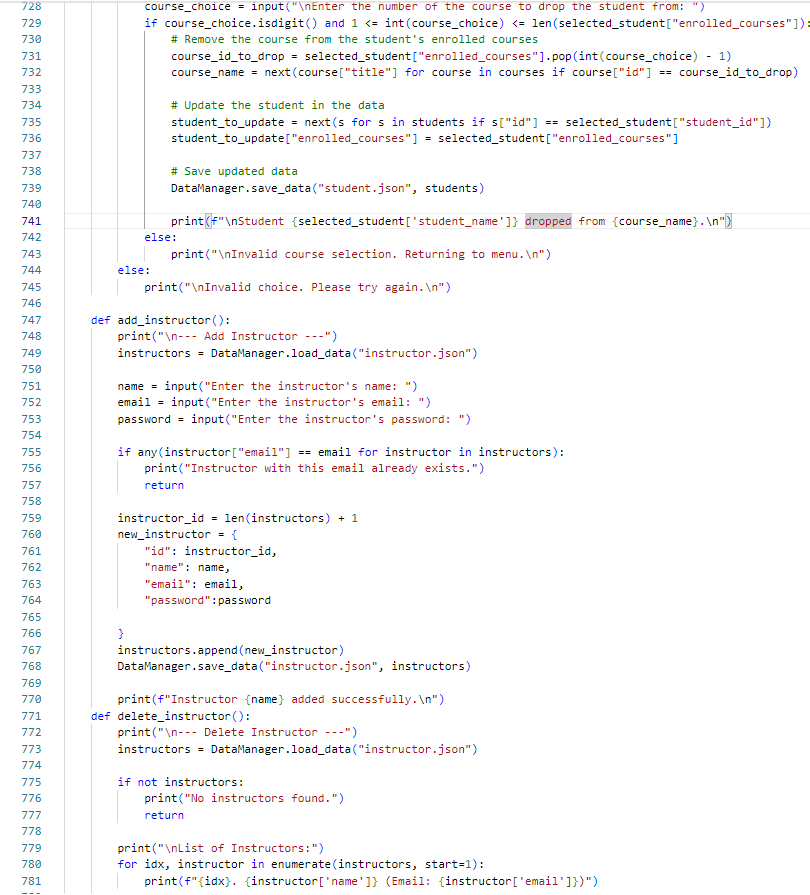
This line retrieves the courses assigned to the instructor using the **PlatformAdmin.get\_courses\_for\_instructor(instructor["id"])** method. The instructor's ID is passed as a parameter to get their courses. If no courses are assigned to the instructor, a message is printed ("You are not teaching any courses."), and the function exits early **(return).** If the instructor is teaching courses, this part of the code displays the list of courses they are teaching. The courses are enumerated and displayed with their index, so the instructor can select one by number. The instructor is prompted to enter a number corresponding to the course they want to view attendance for. If the input is invalid (not a number or out of range), an error message is displayed, and the function exits early. After the instructor selects a course, the subjects within that course are displayed. Each subject is listed with an index. The instructor is prompted to choose a subject within the selected course by entering a number corresponding to the subject. The system loads the attendance records from the "**attendance.json**" file. If there are no attendance records for the selected subject and course, the instructor is informed with a message saying no records were found, and the function exits early. If attendance records are found, they are displayed in a list format. Each record shows the date, the student’s name, and their attendance status (Present/Absent). The **Grade** class represents the grade of a student for a specific assignment. It has three attributes: **assignment\_id**, **student\_id** and **grade**.

This section displays a list of the courses that the student is enrolled in. The courses are retrieved from the **enrolled\_courses** list, and each course is listed with a number corresponding to its index. The student is prompted to select a course by entering a number corresponding to a course in the list. The input is checked to ensure it’s a valid number within the available range of courses. After the student selects a course, the system shows the list of subjects within that course. This is done by accessing the **subjects** attribute of the selected course. The student is prompted to select a subject from the available list in the chosen course. The input is validated to ensure the selection is within the available subjects for that course. The student is asked to enter their name and the date for which they are marking attendance. The student is prompted to choose their attendance status. The options are "Present" or "Absent". The status is assigned based on the student's input. The system loads the current attendance records from the **attendance.json** file. A new attendance record is created with details about the student, the course, the subject, the date, and the attendance status (either "Present" or "Absent"). The new attendance record is appended to the **attendance\_data** list, and the updated list is saved back to the **attendance.json** file. A confirmation message is displayed to the student, confirming that their attendance has been successfully recorded for the selected subject and date. The method is intended for instructors to view attendance records for the subjects they teach. The method begins by printing the header: **--- View Attendance ---.**

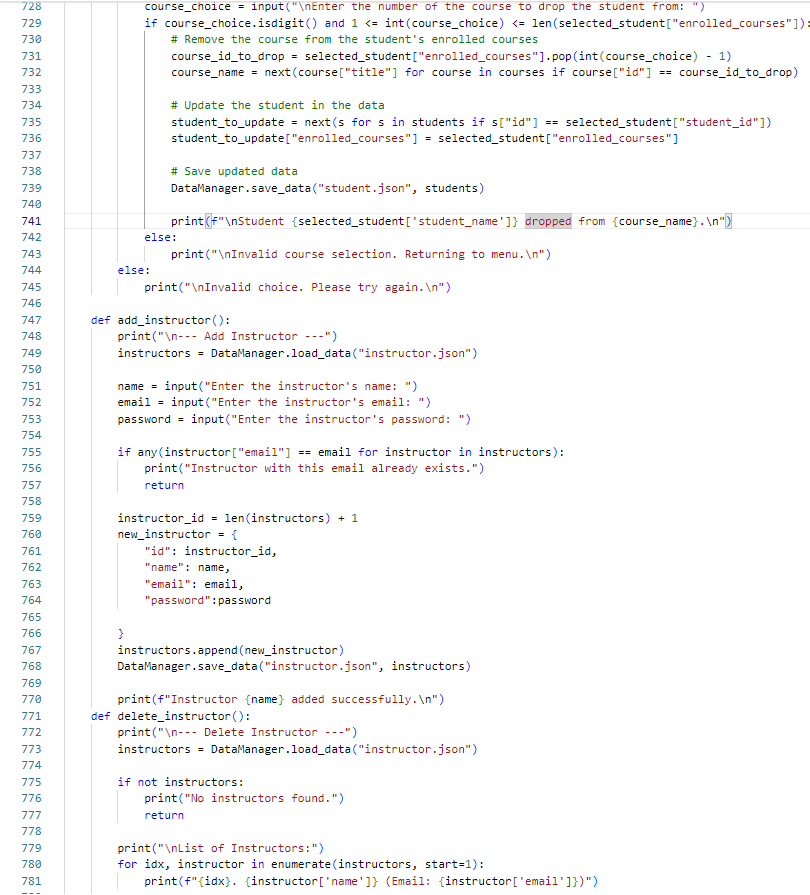
****

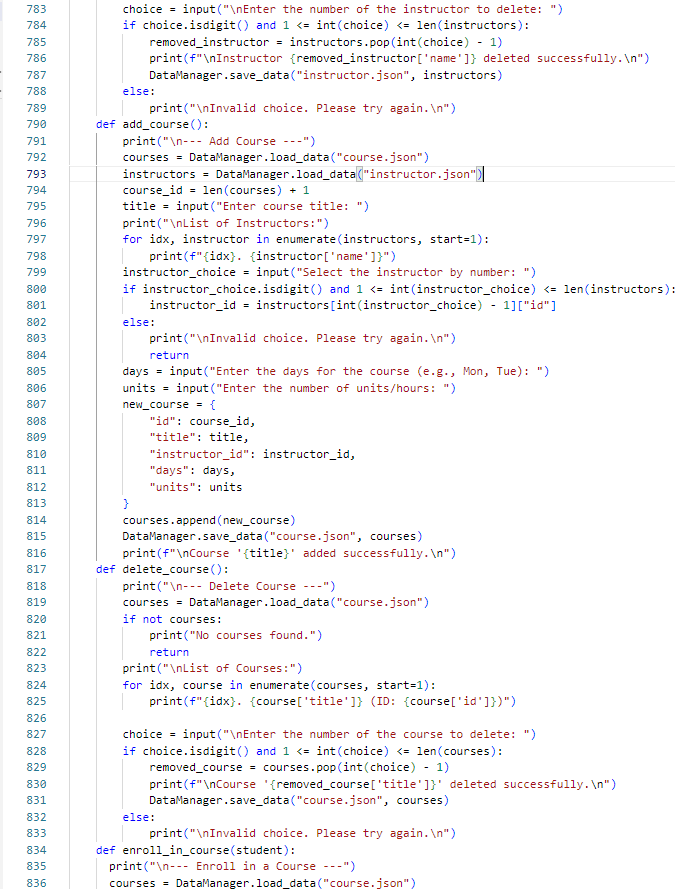
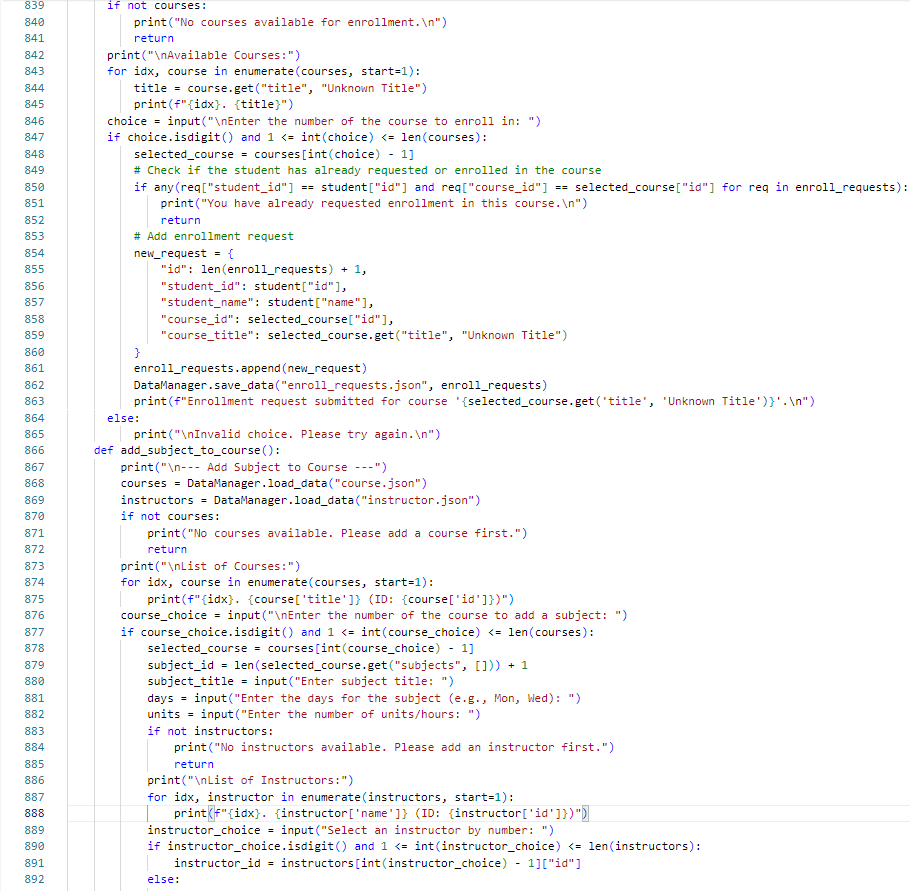
This **Schedule** class is responsible for managing the schedule for a course and viewing the enrolled courses of a student. ***\_\_init\_\_*** Method: Attributes: **course\_id**, **day** and **time**. **view\_enrolled \_courses** Method: This method displays the courses a student is enrolled in and their related details (such as subjects, days, and units). It excludes assignments from the view. It first checks if the student is enrolled in any courses. The student is enrolled in courses, it loads the courses data using **DataManager .load\_data("course.json").** The **DataManager** class is a utility class that handles loading and saving data from/to JSON files. This method loads data from a specified JSON file. It attempts to open the file in read mode **("r").** If the file exists, it returns the data loaded from the file using **json.load().** If the file does not exist (it raises a **FileNotFoundError**), it returns an empty list. **save\_data** Method: This method saves data to a specified JSON file. It opens the file in write mode **("w").** It writes the data to the file in a human-readable format with an indentation of 4 spaces using **json.dump().** The **DataManager** class is responsible for loading and saving data to and from JSON files, making it easier to work with data stored in external files. The PlatformAdmin class represents an admin user on the platform. Admin can perform various tasks, such as managing courses and users. ***\_\_init\_\_*** Method. Attributes: **id** and **name.** The **PlatformAdmin** class is essentially a blueprint for creating platform admin objects. Currently, it only stores the admin's ID and name.

**assign\_final\_grades (Instructor's Role)** This function allows an instructor to assign final grades to students for each subject they are teaching. The function starts by loading the course and student data from external files (e.g., **course.json** and **student.json**) using the **DataManager.load\_data()** function. The system looks for the subjects that are assigned to the current instructor (using the **instructor["id"]**)**.** The **assigned\_subjects** list stores tuples containing the course title and the corresponding subject. If no subjects are assigned to the instructor, a message is printed, and the function exits early (**return**). The function iterates through each assigned subject. For each subject, it looks for students who are enrolled in the corresponding course. The grade is saved in the student’s **grades** attribute, using the **subject ID** as the key. After assigning grades for all students in all subjects, the updated **students’** data is saved back to the **student.json** file. A confirmation message is displayed to indicate that the grades have been successfully assigned. **view\_final\_grade (Student's Role)** This function allows a student to view their final grades for the subjects they are enrolled in. The function starts by loading the student and course data. The system searches for the logged-in student’s data based on the student's ID (**student["id"]).**If the student data is not found, an error message is displayed, and the function exits early. The function checks if the student has any grades. If no grades are available, it informs the student and exits. The function loops through all courses and checks if the student is enrolled in any of them (based on the course ID).

****

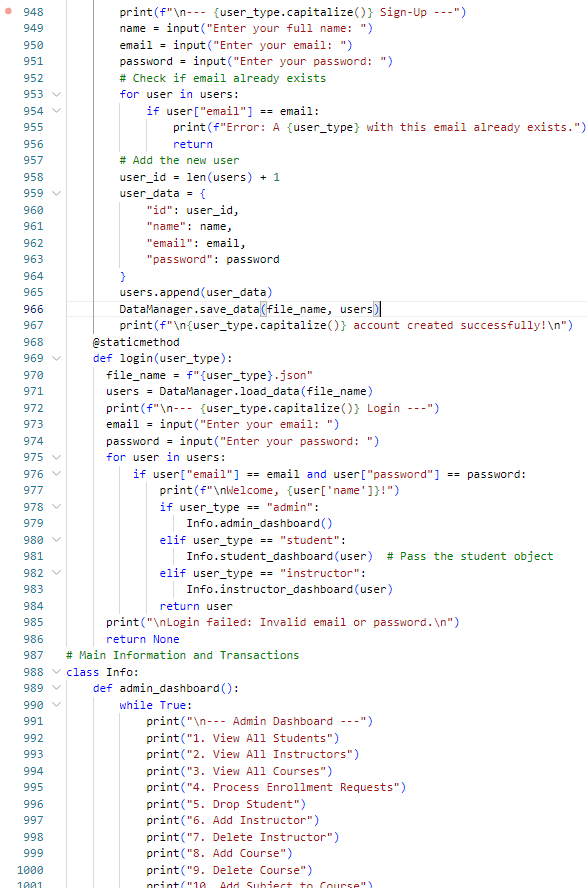
This code extends the administrative functionality for managing students and instructors in a system. Dropping a Student from a Course, this part allows an administrator to drop a student from a selected course. The user is prompted to enter the number of the course from which to drop the student. If the input is valid (i.e., it's a number within the range of enrolled courses), the course is removed from the student's **enrolled\_courses** list. The course ID is retrieved using pop() to remove the course from the list. The corresponding course name is found by matching the **course\_id** in the courses list. The student's enrollment is updated, and the student's data is saved back into **student.json**. A message is displayed confirming that the student has been dropped from the course. If the input is invalid (either not a number or out of range), an error message is displayed. Adding an instructor, this function allows an administrator to add a new instructor to the system. Before adding the new instructor, it checks if the email already exists in the list of existing instructors by searching through the instructor’s data. Adds a new instructor to the system after checking for duplicate emails. Creates a new instructor object and saves it to **instructor.json**. Deleting an instructor, this function allows an administrator to delete an instructor from the system. Displays the list of instructors. To complete the **delete\_instructor()** function, you'll need to add logic to prompt the admin to choose an instructor to delete, remove the instructor from the list, and save the updated list.

This code appears to handle the management of students and their course enrollments, including functions to retrieve subjects from courses, get the courses assigned to an instructor, and drop students from their courses. The **get\_subjects(course\_id)** This function retrieves all subjects for a given course identified by **course\_id.** It loads the data from the **course.json** file using **DataManager.load\_data().** It iterates through each course in the list. If a course with a matching **id** is found, it returns the list of subjects associated with that course. If no course with the given **course\_id** is found, it returns an empty list. The **get\_courses\_for\_Instructor (instructor\_id).** This function retrieves all courses assigned to a particular instructor, based on the **instructor\_id.** It loads the list of courses from the **course.json** file. It filters the courses to include only those where at least one subject has a matching **instructor\_id**. It checks each subject in each course to see if the **instructor\_id** matches the given **instructor\_id**. If so, the course is included in the result. It returns the list of courses assigned to the instructor. The **drop\_student()** This function allows an admin (or a similar role) to drop a student from their enrolled courses. Displays a list of enrolled students and allows the user to select a student to drop from their courses. After displaying the courses, you would need to implement the functionality that actually drops the student by removing the selected course from the student's **enrolled\_courses** and updating the data file. Consider adding error handling in case the user selects a student or course incorrectly. 

****

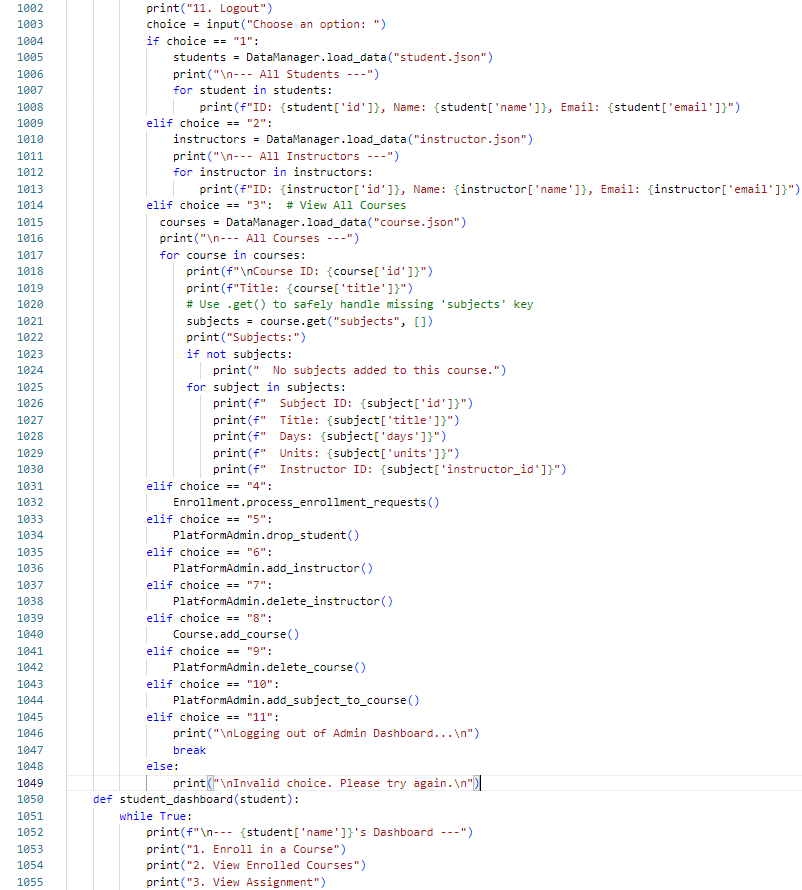
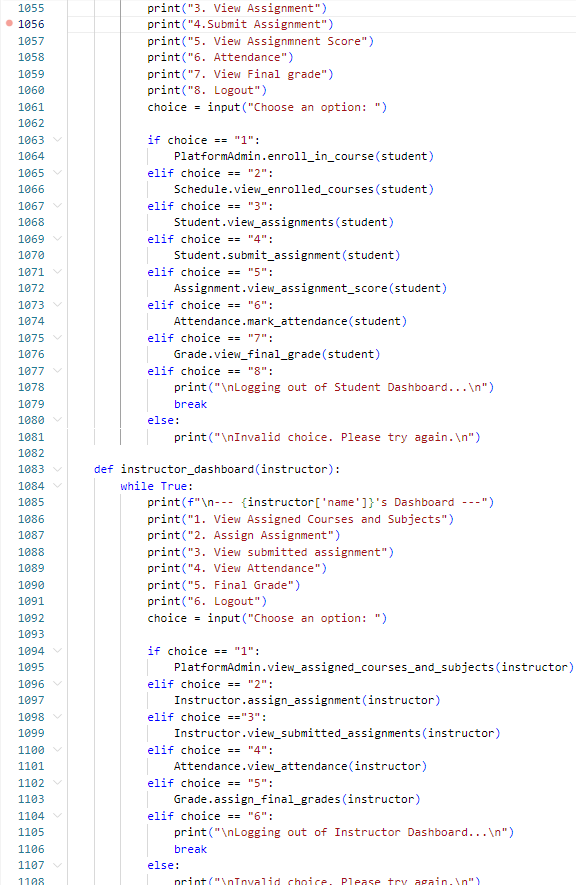
This code handles the process of enrolling a student in a course, submitting enrollment requests, and adding subjects to existing courses. Enrolling a Student in a Course, this section of the code allows a student to submit an enrollment request for a course. If no courses are available, the function prints a message and exits. If courses are available, the code prints a list of them with the title and index number. Before submitting the enrollment request, the code checks if the student has already requested enrollment in the selected course. It uses a list comprehension to search through **enroll\_requests** and checks for any request that matches the student’s ID and the course ID. If no existing request is found, a new enrollment request is created with a unique ID, the student's details, and the selected course's details. If the input is invalid (either not a number or out of range), an error message is displayed. This function allows a student to submit an enrollment request for a course, checking for prior requests and saving the new request to the system. Adding a Subject to a Course, this section of the code adds a subject to an existing course. The function first loads the available courses and instructors. If courses are available, it displays the list of courses with their titles and IDs. The admin or authorized user is prompted to enter the number of the course to which they want to add a subject. If the course selection is valid, the system prepares to add a subject to the selected course. If no instructors exist, a message is displayed, and the function exits. An admin or authorized user to add a subject to an existing course, selecting an instructor and specifying subject details like title, days, and units.

This code is focused on adding, deleting, and enrolling in courses, as well as managing instructors. Adding a Course, this function allows an admin to add a new course to the system. The function first loads the existing courses and instructors from their respective JSON files. It assigns a new course ID based on the number of existing courses Admin can add a new course by entering the course title, selecting an instructor, and providing additional course details (days and units). Deleting a Course, this function allows an admin to delete a course from the system. Admin can delete an existing course by selecting it from a list. Enrolling a Student in a Course, this function starts the process of enrolling a student in a course. Starts the enrollment process for a student to enroll in a course. The logic for actually enrolling the student is not fully implemented in this part of the code, but it would typically involve: Displaying available courses for the student to choose from. Checking if the student meets prerequisites or has any pending enrollment requests. Adding the student to the selected course and updating both the student’s and the course's data. The full logic isn't shown, but it would involve selecting a course and adding the student to it. To fully implement student enrollment, you would need to add more logic for handling course selections, validating prerequisites (if any), and updating both the student’s and course's data.

****

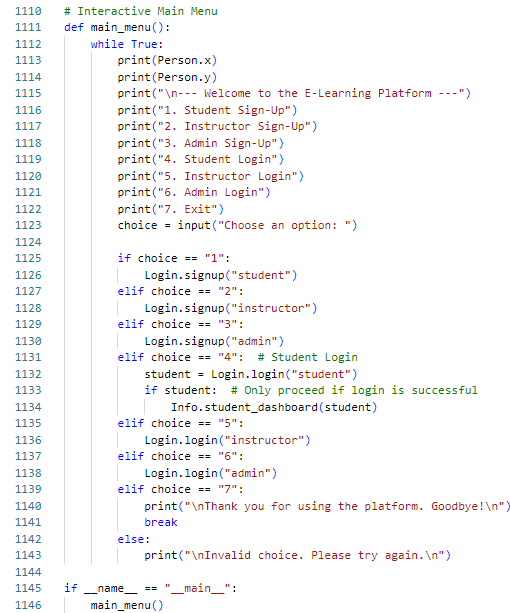
This code involves two main parts: the user authentication system (signup and login) and the **admin\_dashboard** method that provides administrative functionalities. **Sign-Up Method,** the **signup** method allows a new user (either a student, instructor, or admin) to create an account. The code checks if the provided email already exists in the users list. If the email is already in use, it displays an error message and returns, stopping the signup process. Allows users to create a new account (student, instructor, or admin) and checks for email duplicates before adding the user. **Login Method,** the l**ogin** method allows users (admin, student, or instructor) to log in using their email and password. If credentials match, the appropriate dashboard is shown based on the **user\_type**. Authenticates users by checking email and password, and redirects them to the appropriate dashboard based on their user type (admin, student, instructor).**Admin Dashboard Method,** the **admin\_dashboard** method provides an interface for the admin to manage students, instructors, courses, and other operations in the system. Provides a menu of administrative actions for managing users and courses. This setup is part of a user authentication and role-based access control system, where each user type has a different set of privileges. The next steps would involve implementing the actual actions behind each menu option in the admin dashboard, such as viewing users, adding courses, or processing enrollment requests.

This code includes two primary functions related to course and subject management, and the beginning of a Login class for user authentication. Adding a Subject to a Course, this code adds a new subject to an existing course and saves the updated course data. It performs several steps to ensure that the subject is correctly added and the course data is updated. Viewing Assigned Courses and Subjects for an Instructor, this function displays the courses and subjects that are assigned to a specific instructor and the list of students enrolled in those courses. It is helpful for instructors to see their courses, the subjects they are teaching, and the students who are enrolled. **Login** Class and Signup Method, this is the start of a **Login** class which will manage user authentication. The **signup** method is defined as a static method that creates a new user (either a student, instructor, or another type) based on **the user\_type** passed to it. This allows the system to load the appropriate user data (students, instructors, etc.) from the respective JSON file. The method uses the **DataManager.load\_data(file\_name)** function to load the data of the specified user type (e.g., student or instructor) from the corresponding JSON file. The **DataManager** class is responsible for loading and saving data to and from JSON files, although its methods (**load\_data** and **save\_data**) aren't shown in this snippet. The **add\_subject\_to\_**course function ensures that subjects can be added to courses, and it provides input validation for selecting the course and instructor.

****

This section of the code outlines the **Student Dashboard** and **Instructor Dashboard**, providing specific actions for each user role. These dashboards manage tasks relevant to students and instructors, like course enrollment, assignments, attendance, and grades. The **Student Dashboard** is designed for students to manage their learning journey. It provides actions like enrolling in courses, managing assignments, and viewing grades. **Student Dashboard,** this dashboard is customized for students to access and manage their learning activities. Student can perform based on their selected choice: **Enroll in a Course, View Enrolled Courses, View Assignments, Submit Assignment, View Assignment Score, Attendance, View Final Grade** and **Logout**. The **Instructor Dashboard** enables instructors to manage courses, assignments, attendance, and grades. It’s built to help instructors track their responsibilities efficiently. **Instructor Dashboard, t**his dashboard provides tools for instructors to manage courses, assignments, attendance, and grades. Instructor can perform based on their selected choice: **View Assigned Courses**, **Subjects**, **Assign Assignment, View Submitted Assignments, View Attendance, Final Grade,** and **Logout** Each dashboard offers features tailored to the user's role. Students focus on accessing their learning materials and grades. Instructors manage their teaching responsibilities. The dashboards call methods from various classes (**PlatformAdmin**, **Instructor**, **Attendance**, **Grade**) to handle specific tasks. Each method likely interacts with underlying data files or a database for persistence. The code uses default cases to handle invalid inputs, prompting users to try again.

This code represents the continuation of the **admin dashboard** and the beginning of the **student dashboard. Admin Dashboard,** this section builds upon the **admin\_dashboard** function discussed earlier and defines actions and admin can perform based on their selected choice: **View All Students, View All Instructors, View All Courses, Process Enrollment Requests, Drop a Student, Add an Instructor, Delete an Instructor, Add a Course, Delete a Course, Add Subject to a Course and Logout.** The admin is provided with full control over the system's data, allowing them to view, add, or remove students, instructors, courses, and subjects.It also integrates processes like enrollment request handling and course/subject management**. Student Dashboard ,** the student dashboard begins here and provides functionality tailored to students. **Personalized Welcome:** The student's name is displayed at the top of the dashboard for a personalized experience. student can perform based on their selected choice: **Enroll in a Course**, **View Enrolled Courses** and **View Assignment.** The student dashboard is designed for individual users, enabling them to enroll in courses, view their current enrollments, and access assignments. Both dashboards aim to provide role-based functionality, ensuring that admins and students have distinct capabilities appropriate to their roles.

****

This is the **Interactive Main Menu** function for an e-learning platform. It acts as the entry point to the application, providing users with various options to sign up, log in, and navigate to role-specific dashboards. The **main\_menu()** function runs an **infinite loop** to repeatedly display a menu until the user chooses to exit. The menu allows: **Sign-up for different roles (Student, Instructor, Admin)**, **Log in for different roles** and **exit the application**. **Student Sign-Up ,** Calls **Login.signup("student").** Opens the sign-up flow specifically for students. **Instructor Sign-Up, Calls Login.signup("instructor").** Opens the sign-up flow specifically for instructors. **Admin Sign-Up,** Calls **Login.signup("admin").** Opens the sign-up flow for administrators. **Student Login,** Calls **Login.login("student").** Prompts the student for their email and password.  **Instructor Login,** Calls **Login.login("instructor").** Prompts the instructor for email and password. **Admin Login,** Calls **Login.login("admin").** Prompts the admin for email and password. **Exit**, Exits the infinite loop. Prints a goodbye message and ends the program. **if \_\_name\_\_ == "\_\_main\_\_":** This ensures that the **main\_menu()** function runs only when the script is executed directly (and not when imported as a module into another script). he main menu acts as the **entry point** to the application, handling the user journey from account creation to specific operations.