# Student Performance Analysis CLI: Project Documentation

# Preface: Course Project Overview

This document serves as the comprehensive project documentation for the "Student Performance Analysis CLI" python-based application, developed as a fulfillment of the submission requirements for **LAUTECH – CSC 202 (INTRODUCTION TO PROGRAMMING LAB II)**. This project demonstrates a practical application of core programming concepts like Object-Oriented Programming (OOP) principles, file handling, robust error management, and modular development. It aims to provide a robust command-line tool for educators and administrators to analyze student academic performance effectively.

Contents

[Student Performance Analysis CLI: Project Documentation 1](#_Toc204873352)

[Preface: Course Project Overview 2](#_Toc204873353)

[1. Members 0f Group 38 5](#_Toc204873354)

[2. Project Overview and Goals 6](#_Toc204873355)

[3. Requirements Specification 7](#_Toc204873356)

[3.1. Project Requirements Fulfillment 7](#_Toc204873357)

[3.2. Technical Requirements Fulfillment 7](#_Toc204873358)

[4. User Guide 9](#_Toc204873359)

[4.1. Installation and Setup 9](#_Toc204873360)

[4.2. Running the Application 9](#_Toc204873361)

[4.3. Main Menu Options 10](#_Toc204873362)

[4.3.1. Load Another Class/Department Data (Option 1) 10](#_Toc204873363)

[4.3.3. View Current Class Summary (Option 3) 10](#_Toc204873364)

[4.3.5. View Current Class All Students (Option 5) 10](#_Toc204873365)

[4.3.6. View Student Learning Progress (Current Class) (Option 6) 11](#_Toc204873366)

[4.3.7. Compare Loaded Classes (Option 7) 11](#_Toc204873367)

[4.3.8. Save Current Class Data (Option 8) 11](#_Toc204873368)

[4.3.9. Exit (Option 9) 11](#_Toc204873369)

[4.4. Navigating and Cancelling Operations 11](#_Toc204873370)

[5. Technical Documentation 12](#_Toc204873371)

[5.1. Object-Oriented Programming (OOP) and Class Hierarchy 12](#_Toc204873372)

[5.2. Module Breakdown 13](#_Toc204873373)

[5.3. Data Structures and Persistence 13](#_Toc204873374)

[5.4. Error Handling and Input Validation 13](#_Toc204873375)

[5.5. Unit Testing Strategy 14](#_Toc204873376)

[5.6. Code Implementation 15](#_Toc204873377)

[5.6.1. Main.py 15](#_Toc204873378)

[5.6.1. models.py 24](#_Toc204873379)

[5.6.3 grade\_calculator.py 29](#_Toc204873380)

[5.6.4 report\_generator.py 37](#_Toc204873381)

[5.7 Successful Implementation with testing 42](#_Toc204873382)

[5.7.1. test\_student.py 43](#_Toc204873383)

[5.7.2. Output of the test 52](#_Toc204873384)

[5.7.3. Performance Tool Summary sample 53](#_Toc204873385)

[6. Conclusion 54](#_Toc204873386)

# Members of Group 38

* **Oluwafemi Emmanuel Oluwaseyin - 2023007009**
* **Olayinka Promise Adebayo - 2023003993**
* **Olajide Zainab Adebare - 2023007253**
* **Akinpeju Oluwadarasimi Esther – 2023006836**
* **Abdur – Rahman Lateifah Titilayo - 2023003273**

# 2. Project Overview and Goals

The Student Performance Analysis CLI is a Python-based command-line application designed to assist educational institutions in managing and analyzing student academic data. The primary goal is to transform raw academic scores from CSV files into actionable insights, facilitating better understanding of student achievement and informing decisions for academic improvement.

**Key Goals of the Project:**

* **Data Management:** Efficiently process and store academic data from various CSV sources, including multi-year records.
* **Performance Metrics Calculation:** Accurately compute individual student GPAs, average scores, and class-level performance metrics.
* **Learning Progress Tracking:** Provide insights into individual student academic progression over different academic periods (e.g., 100L, 200L, 300L).
* **Achievement Pattern Analysis:** Identify and visualize grade distributions across subjects and classes.
* **Class Comparison:** Enable comparative analysis between different academic departments or classes to highlight relative strengths and weaknesses.
* **Recommendations for Improvement:** Generate basic, data-driven recommendations for students based on their performance.
* **User-Friendly Interface:** Offer an intuitive Command Line Interface (CLI) with robust input validation and navigation options.
* **Data Persistence:** Allow users to save modified or newly entered data back to CSV files.
* **Modular and Maintainable Code:** Adhere to Object-Oriented Programming (OOP) principles for a scalable and easily maintainable codebase.

# 3. Requirements Specification

This section details how the project fulfills the specified requirements.

## 3.1. Project Requirements Fulfillment

* **Implemented as Command Line Interface (CLI) Application:** The entire application operates exclusively through the command line, with no Graphical User Interface (GUI) or web components. All interactions are text-based.
* **Demonstrates Understanding of Course Modules:** The project extensively uses OOP principles (classes, objects, inheritance, polymorphism, encapsulation), file I/O (CSV handling), data structures (dictionaries, lists, defaultdict), error handling (try-except), and modular programming.
* **Includes Proper Error Handling, File Operations, and Object-Oriented Design:**
  + **Error Handling:** Comprehensive try-except blocks are used for file operations (FileNotFoundError, ValueError during score conversion), invalid user input, and general unexpected errors. Users are provided with informative messages and options to recover or retry.
  + **File Operations:** The application reads student data from CSV files and can write modified or new data back to CSV files, ensuring data persistence.
  + **Object-Oriented Programming (OOP):** The system is designed with clear responsibilities for each class (Person, Student, GradeCalculator, ReportGenerator), promoting modularity, reusability, and maintainability.
* **Solves a Meaningful Real-World Problem:** The application addresses the practical need for academic performance analysis in educational settings, providing tools for tracking student progress, identifying areas for improvement, and comparing class performance.
* **Requires Research Beyond Classroom Materials:** (Self-assessment for the student) The implementation of dynamic CSV header generation for saving, the structured multi-year data handling, and the detailed error recovery mechanisms often require going beyond basic classroom examples.
* **Includes Comprehensive Documentation and User Manual:** This document itself serves as the comprehensive documentation and user manual.
* **Implements Robust Input Validation and Error Recovery:** User inputs are validated for correctness (e.g., numeric input for scores, valid menu choices). The get\_user\_input helper function allows users to "back" out of operations, enhancing error recovery and user experience.

## 3.2. Technical Requirements Fulfillment

* **Uses Python 3.8+ with Only Standard Library Modules:** The project exclusively relies on Python's standard library modules (csv, os, statistics, unittest, collections.defaultdict, sys). No external libraries are used.
* **Implements Proper Class Hierarchies with Inheritance and Polymorphism:**
  + **Inheritance:** The Student class inherits from a Person base class, demonstrating a clear "is-a" relationship.
  + **Polymorphism:** The display\_info() method is defined in both Person and Student classes, with Student overriding the base class method to provide specialized output. This allows different object types to respond to the same method call in a type-specific way.
* **Includes File Handling for Data Persistence:** Data is loaded from CSV files and can be saved back to CSV files, ensuring that manually added data or modifications are not lost.
* **Implements Comprehensive Error Handling:** As detailed in 3.1, error handling is integrated throughout the application, from file operations to user input.
* **Creates Modular, Well-Documented Code:** The project is logically divided into separate Python modules (models.py, grade\_calculator.py, report\_generator.py, main.py), each with a specific role. All classes, methods, and complex logic are accompanied by clear docstrings and inline comments.
* **Includes at Least 5 Different Modules from Python's Standard Library:** The project uses csv, os, statistics, unittest, collections (specifically defaultdict), and sys, exceeding the minimum requirement.
* **Provides Unit Tests for Critical Functions:** A dedicated test\_student.py module contains unit tests for core functionalities of Person, Student, GradeCalculator, and ReportGenerator classes, ensuring correctness and reliability

# 4. User Guide

This section provides instructions on how to set up, run, and interact with the Student Performance Analysis CLI application.

## 4.1. Installation and Setup

1. **Python Installation:** Ensure you have Python 3.8 or higher installed on your system. You can download it from [python.org](https://www.python.org/downloads/).
2. **Project Files:** Place all the Python script files (main.py, models.py, grade\_calculator.py, report\_generator.py, test\_student.py) and your CSV data files (e.g., physics.csv, english.csv, etc.) in the same directory.

## 4.2. Running the Application

1. **Open Terminal/Command Prompt:** Navigate to the directory where you saved your project files using your terminal or command prompt.
2. **Execute main.py:** Run the application using the command:
3. python main.py
4. **Initial Data Load:** The application will first prompt you to load an initial class/department data.
   * **Enter a name for this class/department (e.g., Physics, English):** Type a descriptive name for the dataset you are about to load (e.g., Physics Department).
   * **File Selection Options:** You will then be presented with options to provide the CSV file:
     + **1. Select an existing CSV file by number:** If you see your CSV files listed (e.g., 1. physics.csv), type the corresponding number (e.g., 1).
     + **2. Enter a full path to a CSV file:** If your file is not in the current directory or not listed, type its full path (e.g., data/physics.csv or C:\Users\YourName\Documents\physics.csv).
     + **3. Create a new empty CSV file:** Type new to create a new CSV file. You will be prompted for a filename. This is useful for starting a new class from scratch.
     + **4. Start with empty data (for manual entry):** Leave the prompt blank and press Enter. This will create a new class in memory without loading from a file, allowing you to manually add student records.
     + **(Type 'back' to cancel):** At any point during file selection, you can type back to return to the previous step (e.g., re-enter the department name).

## 4.3. Main Menu Options

Once an initial class is loaded, the main menu will be displayed. The Current active class: will show the name of the class you are currently working with.

### 4.3.1. Load Another Class/Department Data (Option 1)

* Allows you to load additional CSV files, each representing a different class or department.
* You will be prompted for a name for the new class and then presented with the same file selection options as the initial load.
* The newly loaded class will become the Current active class.

**4.3.2.** Add/Update Student Record (Current Class) (Option 2)

* Allows you to manually add a new student record or update an existing one within the Current active class.
* You will be prompted for:
  + **Student's full name:** (e.g., Alice Brown)
  + **Academic year:** (e.g., 100L, 200L, 300L)
  + **Subject:Score:** You will enter subjects and their scores one by one (e.g., Math:85). Type done when you have finished entering scores for that student and year.
* The data is updated in memory. Remember to use Option 8 to save these changes to a file.

### 4.3.3. View Current Class Summary (Option 3)

* Displays an overview of the Current active class, including:
  + Total number of students.
  + Overall class average score (based on the latest academic period for each student).
  + Grade distribution (counts of A, B, C, D, E, F grades across all subjects for the latest period).

**4.3.4. View Current Class Top Students (Option 4)**

* Displays a list of students from the Current active class ranked by their GPA (from their latest academic period).
* You will be prompted to enter the number of top students to display (default is 5).

### 4.3.5. View Current Class All Students (Option 5)

* Displays a comprehensive list of all students in the Current active class, including their latest GPA, average score, and personalized recommendations for improvement. Students are listed in ranked order.

### 4.3.6. View Student Learning Progress (Current Class) (Option 6)

* Allows you to view the academic progress of a specific student over different academic periods.
* You will be prompted to enter the full name of the student.
* The report will show their average score trend across years and their individual subject scores for each year.

### 4.3.7. Compare Loaded Classes (Option 7)

* Compares key performance metrics (total students, class average, grade distribution) across all classes/departments you have loaded into the application.
* Requires at least two classes to be loaded.

### 4.3.8. Save Current Class Data (Option 8)

* Saves all in-memory student data for the Current active class back to its associated CSV file.
* If the class was initially created for manual entry (no CSV loaded), you will be prompted to enter a new filename to save the data.
* **Important:** Always save your data after making manual additions or modifications to ensure persistence.

### 4.3.9. Exit (Option 9)

* Exits the application.

### 4.4. Navigating and Cancelling Operations

* **back Option:** For most prompts that require user input (e.g., student name, academic year, file path), you can type back (case-insensitive) to cancel the current operation and return to the previous menu or step. This is indicated in the prompt itself (e.g., (Type 'back' to cancel) >).
* **Invalid Input:** If you enter invalid input (e.g., non-numeric for a choice, incorrect format for scores), the application will display an error message and usually allow you to retry the input.

# 5. Technical Documentation

## 5.1. Object-Oriented Programming (OOP) and Class Hierarchy

The application is built upon a robust Object-Oriented Programming, emphasizing clear separation of concerns, modularity, and reusability.

* **Person Class (Base Class):**
  + **Responsibility:** Represents a generic individual with a name. Provides a basic display\_info() method.
  + **Purpose:** Serves as the base class for Student, demonstrating inheritance.
* **Student Class (Derived Class):**
  + **Responsibility:** Extends Person to represent a student. Manages individual student academic data, including scores across multiple academic periods (all\_scores). Calculates individual average scores, GPAs, identifies weak subjects, generates improvement recommendations, and tracks performance over time.
  + **OOP Principles:**
    - **Inheritance:** Inherits name from Person.
    - **Encapsulation:** Bundles student data (name, all\_scores, average, gpa) with methods that operate on that data.
    - **Polymorphism:** Overrides display\_info() from Person to provide student-specific details.
* **GradeCalculator Class:**
  + **Responsibility:** Manages a collection of Student objects for a specific class/department. Handles loading student data from CSV files (including multi-year data), adding/updating student records in memory, and saving data back to CSV. Performs class-level calculations such as overall class average and grade distribution, and ranks students.
  + **OOP Principles:**
    - **Composition:** "Has-a" relationship with Student objects (manages a list of Student instances).
    - **Error Handling:** Implements robust error handling for file operations and data parsing.
* **ReportGenerator Class:**
  + **Responsibility:** Takes a GradeCalculator instance and generates various formatted academic reports for display to the user. This includes class summaries, top student lists, all student records (with recommendations), individual student learning progress, and comparison reports across multiple classes.
  + **OOP Principles:**
    - **Composition:** "Has-a" relationship with a GradeCalculator instance.
    - **Delegation:** Delegates specific calculations and data retrieval to the GradeCalculator and Student objects.
    - **Static Method:** The compare\_classes method is a static method, operating on a collection of GradeCalculator instances without needing a specific ReportGenerator instance.

## 5.2. Module Breakdown

The project is structured into four distinct Python modules:

* **models.py:** Defines the Person and Student classes, encapsulating the fundamental data structures and core logic related to individual student performance.
* **grade\_calculator.py:** Contains the GradeCalculator class, responsible for data loading, persistence, and class-level analytical operations.
* **report\_generator.py:** Houses the ReportGenerator class, dedicated to formatting and displaying all the various academic reports.
* **main.py:** Serves as the application's entry point, managing the CLI, user interactions, menu navigation, and orchestrating calls to the other modules.

## 5.3. Data Structures and Persistence

* **In-Memory Data:**
  + Student.all\_scores: A nested dictionary {'AcademicPeriod': {'SubjectName': Score}} to store multi-year, multi-subject scores for each student.
  + GradeCalculator.students: A list holding Student objects.
  + GradeCalculator.\_students\_by\_name: A dictionary mapping student names to Student objects for quick lookup.
  + main.loaded\_classes: A dictionary mapping class/department names to GradeCalculator instances, enabling multi-class management.
* **Data Persistence:**
  + Data is loaded from CSV files using the csv module.
  + The GradeCalculator.save\_students() method reconstructs the multi-year student data back into a flat CSV format for saving, ensuring all changes (from manual entry or modifications) are written to disk.

## 5.4. Error Handling and Input Validation

The application implements robust error handling to ensure stability and provide a user-friendly experience:

* **File Operations:** FileNotFoundError is caught during CSV loading, prompting the user to re-enter the path.
* **Data Parsing:** ValueError and TypeError are handled during score conversion from CSV, with warnings printed for invalid data rows without crashing the application.
* **User Input Validation:**
  + Menu choices are validated to be within the allowed range.
  + Numeric inputs (e.g., number of top students) are validated.
  + Scores entered manually are validated to be numeric and within the 0-100 range.
  + Empty inputs for critical fields (names, years) are handled.
* **Error Recovery:** The get\_user\_input helper function allows users to type back to cancel an operation and return to a previous state, preventing frustration from being stuck in a prompt.

## 5.5. Unit Testing Strategy

The project employs a comprehensive unit testing strategy using Python's built-in unittest framework. All critical functions and classes have dedicated test cases to ensure their correctness and reliability.

* **Test Modules:** All unit tests are consolidated into a single test\_student.py file.
* **Test Cases:** Separate unittest.TestCase classes are defined for Person, Student, GradeCalculator, and ReportGenerator.
* **Setup/Teardown:** setUp and tearDown methods are used to create and clean up temporary CSV files for tests involving file operations, ensuring test isolation.
* **Mocking:** The unittest.mock.patch decorator is used to simulate user input (builtins.input) for interactive methods and to suppress print output during tests, allowing for focused assertion on method logic rather than console output.
* **Coverage:** Tests cover:
  + Class initialization and attribute assignment.
  + Calculation logic (GPA, averages, grade points).
  + Data loading from CSV (valid, invalid, missing files).
  + Manual data addition and updates.
  + Data saving to CSV (new and existing files).
  + Learning progress tracking.
  + Weak subject identification and recommendation generation.
  + Class-level statistics (class average, grade distribution, ranking).
  + Class comparison.
  + Edge cases (empty data, invalid inputs).

## 5.6. Code Implementation

The following codes that would be shown in this subsection are the main codes of the file that are used to run the student performance

### 5.6.1. Main.py

# main.py

from grade\_calculator import GradeCalculator

from report\_generator import ReportGenerator

import sys # Import sys for exiting the program gracefully

import os  # Import os for listing files

"""

This is the main entry point for the Student Performance Analysis CLI Application.

It handles user interaction, loads student data (now supporting multi-year data),

and generates various academic reports including class summaries, top students,

all student records, individual student learning progress, class comparison,

and now allows for manual addition and saving of student records.

It features improved robustness with a 'back' option, enhanced user guidance,

and the ability to select CSV files by number or create new ones.

"""

# Define a constant for the prompt symbol for consistency

PROMPT\_SYMBOL = "> "

def show\_menu():

    """

    Displays the main menu options to the user.

    """

    print("\n" + "=" \* 35)

    print("  STUDENT PERFORMANCE ANALYZER")

    print("=" \* 35)

    print("1. Load Another Class/Department Data")

    print("2. Add/Update Student Record (Current Class)")

    print("3. View Current Class Summary")

    print("4. View Current Class Top Students")

    print("5. View Current Class All Students")

    print("6. View Student Learning Progress (Current Class)")

    print("7. Compare Loaded Classes")

    print("8. Save Current Class Data") # New option for saving

    print("9. Exit") # Updated exit option

    print("=" \* 35)

def get\_user\_input(prompt: str, allow\_back: bool = True) -> str | None:

    """

    Helper function to get user input with a consistent prompt symbol and

    an option to go back.

    Args:

        prompt (str): The message to display to the user.

        allow\_back (bool): If True, informs the user they can type 'back' to cancel.

    Returns:

        str | None: The user's input string, or None if 'back' was entered.

    """

    if allow\_back:

        full\_prompt = f"{prompt} (Type 'back' to cancel) {PROMPT\_SYMBOL}"

    else:

        full\_prompt = f"{prompt} {PROMPT\_SYMBOL}"

    user\_input = input(full\_prompt).strip()

    if allow\_back and user\_input.lower() == 'back':

        return None

    return user\_input

def list\_csv\_files(directory: str = '.') -> list[str]:

    """

    Lists all .csv files in the specified directory.

    Args:

        directory (str): The directory to search in. Defaults to the current directory.

    Returns:

        list[str]: A list of CSV filenames.

    """

    csv\_files = [f for f in os.listdir(directory) if f.endswith('.csv') and os.path.isfile(os.path.join(directory, f))]

    return sorted(csv\_files) # Sort for consistent numbering

def select\_filepath\_or\_create(dept\_name: str) -> str | None:

    """

    Guides the user to select an existing CSV file by number, enter a path,

    create a new empty CSV file, or start with empty data.

    Args:

        dept\_name (str): The name of the department/class being loaded, for context.

    Returns:

        str | None: The selected/created file path, an empty string if manual entry

                    is chosen, or None if the user cancels.

    """

    csv\_files = list\_csv\_files()

    while True:

        print("\n--- File Selection Options for", dept\_name, "---")

        print("1. Select an existing CSV file by number from the list below.")

        print("2. Enter a full path to a CSV file.")

        print("3. Create a new empty CSV file.")

        print("4. Start with empty data (for manual entry).")

        print("---------------------------------------")

        if csv\_files:

            print("Available CSV files in current directory:")

            for i, file in enumerate(csv\_files):

                print(f"  {i+1}. {file}")

        else:

            print("  No CSV files found in current directory.")

        action\_choice = get\_user\_input("Choose an option (1-4)", allow\_back=True)

        if action\_choice is None: # User typed 'back'

            return None

        if not action\_choice.isdigit() or not (1 <= int(action\_choice) <= 4):

            print("Invalid option. Please choose a number between 1 and 4.")

            continue

        action\_choice = int(action\_choice)

        if action\_choice == 1: # Select from listed files

            if not csv\_files:

                print("No CSV files available to select by number. Please choose another option.")

                continue

            file\_num\_str = get\_user\_input("Enter the number of the CSV file to select")

            if file\_num\_str is None: continue

            if not file\_num\_str.isdigit():

                print("Invalid input. Please enter a number.")

                continue

            idx = int(file\_num\_str) - 1

            if 0 <= idx < len(csv\_files):

                return csv\_files[idx]

            else:

                print("Invalid number. Please select a number from the list.")

                continue

        elif action\_choice == 2: # Enter a full path

            filepath = get\_user\_input("Enter the full path to the CSV file")

            if filepath is None: continue

            if not filepath:

                print("File path cannot be empty. Please try again.")

                continue

            return filepath

        elif action\_choice == 3: # Create a new empty CSV file

            new\_filename = get\_user\_input("Enter the name for the new CSV file (e.g., my\_new\_class.csv)")

            if new\_filename is None: continue

            if not new\_filename:

                print("Filename cannot be empty. Aborting file creation.")

                continue

            if not new\_filename.lower().endswith('.csv'):

                new\_filename += '.csv'

            try:

                with open(new\_filename, 'w', newline='', encoding='utf-8') as f:

                    f.write("Name,Department,Year\n") # Basic headers

                print(f"Successfully created new empty CSV file: '{new\_filename}'")

                return new\_filename

            except Exception as e:

                print(f"Error creating file '{new\_filename}': {e}. Please try again.")

                continue

        elif action\_choice == 4: # Start with empty data (manual entry)

            return "" # Empty string signals manual entry

def main():

    """

    The main function that runs the student performance analysis application.

    It prompts the user for CSV files, loads data, and allows the user

    to navigate through different report options, including learning progress,

    class comparison, and manual data entry.

    """

    print("Welcome to the Student Performance Analyzer!")

    loaded\_classes = {}

    current\_class\_name = None

    print("\n--- Initial Class Data Load ---")

    print("You need to load at least one class to start.")

    while not loaded\_classes:

        dept\_name = get\_user\_input("Enter a name for this class/department (e.g., Physics, English)", allow\_back=False)

        if not dept\_name:

            print("Class/Department name cannot be empty. Please try again.")

            continue

        filepath = select\_filepath\_or\_create(dept\_name)

        if filepath is None: # User typed 'back' during file selection

            print("Operation cancelled. Please load a class to continue.")

            continue # Stay in loop to prompt for file again

        # If filepath is empty, create a calculator for manual entry

        if not filepath: # This means user chose to start with empty data

            calculator = GradeCalculator(filepath=None, department\_name=dept\_name) # Pass department\_name

            loaded\_classes[dept\_name] = calculator

            current\_class\_name = dept\_name

            print(f"Initialized '{dept\_name}' for manual data entry.")

            break # Exit initial load loop as a class is now 'loaded'

        else: # User provided a file path (either by typing or selecting a number)

            calculator = GradeCalculator(filepath=filepath, department\_name=dept\_name) # Pass department\_name

            try:

                calculator.load\_students()

                if not calculator.students:

                    print(f"No valid student records found in '{filepath}'. Please check the file content or try another file.")

                    continue

                else:

                    loaded\_classes[dept\_name] = calculator

                    current\_class\_name = dept\_name

                    print(f"Successfully loaded '{dept\_name}' data.")

            except FileNotFoundError as e:

                print(f"{e}")

                continue

            except Exception as e:

                print(f"An unexpected error occurred during file loading: {e}")

                continue

        break # Break from initial load loop if successful

    # Main application loop

    while True:

        show\_menu()

        if current\_class\_name:

            print(f"Current active class: {current\_class\_name}")

        else:

            print("Error: No active class selected (this should not happen after initial load).")

        choice = get\_user\_input("Choose an option", allow\_back=False) # Main menu doesn't have 'back'

        current\_calculator = loaded\_classes.get(current\_class\_name)

        if current\_calculator:

            report = ReportGenerator(current\_calculator)

        else:

            print("Error: No current class selected or loaded. Please load a class first (Option 1).")

            continue

        if choice == '1': # Load Another Class/Department Data

            dept\_name = get\_user\_input("Enter a name for this new class/department (e.g., Chemistry, Arts)")

            if dept\_name is None: continue # User typed 'back'

            if not dept\_name:

                print("Class/Department name cannot be empty. Aborting load.")

                continue

            if dept\_name in loaded\_classes:

                print(f"Class '{dept\_name}' already loaded. Overwriting existing data.")

            filepath = select\_filepath\_or\_create(dept\_name)

            if filepath is None: continue # User typed 'back'

            if not filepath: # User chose to start with empty data

                new\_calculator = GradeCalculator(filepath=None, department\_name=dept\_name) # Pass department\_name

                loaded\_classes[dept\_name] = new\_calculator

                current\_class\_name = dept\_name

                print(f"Initialized '{dept\_name}' for manual data entry.")

            else: # User provided a file path (typed or selected by number)

                new\_calculator = GradeCalculator(filepath=filepath, department\_name=dept\_name) # Pass department\_name

                try:

                    new\_calculator.load\_students()

                    if not new\_calculator.students:

                        print(f"No valid student records found in '{filepath}'. Data for '{dept\_name}' not loaded.")

                    else:

                        loaded\_classes[dept\_name] = new\_calculator

                        current\_class\_name = dept\_name

                        print(f"Successfully loaded and switched to '{dept\_name}' data.")

                except FileNotFoundError as e:

                    print(f"{e}. Data for '{dept\_name}' not loaded.")

                except Exception as e:

                    print(f"An unexpected error occurred during file loading: {e}. Data for '{dept\_name}' not loaded.")

        elif choice == '2': # Add/Update Student Record

            print("\n--- Add/Update Student Record ---")

            student\_name = get\_user\_input("Enter student's full name")

            if student\_name is None: continue

            if not student\_name:

                print("Student name cannot be empty. Aborting.")

                continue

            academic\_year = get\_user\_input("Enter academic year (e.g., 100L, 200L, 300L)")

            if academic\_year is None: continue

            if not academic\_year:

                print("Academic year cannot be empty. Aborting.")

                continue

            scores = {}

            print("Enter subject scores (e.g., Math:85, English:70). Type 'done' when finished.")

            while True:

                subject\_input = get\_user\_input("Subject:Score (or 'done')", allow\_back=False)

                if subject\_input.lower() == 'done':

                    break

                if ':' in subject\_input:

                    try:

                        subject, score\_str = subject\_input.split(':', 1)

                        score = float(score\_str)

                        if not (0 <= score <= 100):

                            print("Score must be between 0 and 100. Please re-enter.")

                            continue

                        scores[subject.strip()] = score

                    except ValueError:

                        print("Invalid score format. Please use 'Subject:Score' (e.g., Math:85).")

                    except Exception as e:

                        print(f"An error occurred: {e}. Please try again.")

                else:

                    print("Invalid format. Please use 'Subject:Score' (e.g., Math:85).")

            if not scores:

                print("No valid scores entered. Record not added/updated.")

                continue

            current\_calculator.add\_student\_record(student\_name, academic\_year, scores)

            print("---------------------------------")

        elif choice == '3': # View Current Class Summary

            report.display\_summary()

        elif choice == '4': # View Current Class Top Students

            while True:

                limit\_str = get\_user\_input("Enter number of top students to display (default 5)")

                if limit\_str is None:

                    break

                if not limit\_str:

                    limit = 5

                else:

                    try:

                        limit = int(limit\_str)

                        if limit <= 0:

                            print("Please enter a positive number.")

                            continue

                        break

                    except ValueError:

                        print("Invalid input. Please enter a number.")

            if limit\_str is not None:

                report.display\_top\_students(limit)

        elif choice == '5': # View Current Class All Students

            report.display\_all\_students()

        elif choice == '6': # View Student Learning Progress (Current Class)

            student\_name = get\_user\_input("Enter the name of the student to view progress")

            if student\_name is None: continue

            report.display\_student\_learning\_progress(student\_name)

        elif choice == '7': # Compare Loaded Classes

            if len(loaded\_classes) < 2:

                print("Please load at least two classes/departments (Option 1) to perform a comparison.")

            else:

                print("\n--- Available Classes for Comparison ---")

                for i, name in enumerate(loaded\_classes.keys(), 1):

                    print(f"{i}. {name}")

                print("---------------------------------------")

                ReportGenerator.compare\_classes(loaded\_classes)

        elif choice == '8': # Save Current Class Data (NEW)

            if current\_calculator.filepath is None and not current\_calculator.students:

                print("This class was created for manual entry but has no data to save yet.")

                print("Please add some student records (Option 2) before saving.")

            else:

                current\_calculator.save\_students()

        elif choice == '9': # Exit

            print("Thank you for using the Student Performance Analyzer. Goodbye!")

            sys.exit(0)

        else:

            print("Invalid option. Please choose a number between 1 and 9.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

### 5.6.1. models.py

# models.py

"""

This module defines the core data models for the student performance analysis system.

It includes a base Person class and a Student class that inherits from Person,

demonstrating Object-Oriented Programming (OOP) principles like inheritance and polymorphism.

It now supports tracking learning progress by storing scores for different academic periods.

"""

class Person:

    """

    A base class representing a generic person.

    It encapsulates basic information common to all individuals.

    """

    def \_\_init\_\_(self, name: str):

        """

        Initializes a new Person object.

        Args:

            name (str): The name of the person.

        """

        self.name = name

    def display\_info(self):

        """

        Displays basic information about the person.

        This method can be overridden by subclasses to provide more specific details.

        """

        return f"Name: {self.name}"

class Student(Person):

    """

    Represents a student, inheriting from the Person class.

    It extends the Person's attributes with academic scores, now supporting

    scores across multiple academic periods (e.g., years).

    Calculates average score and GPA for a specific period or overall.

    Includes methods to identify weak subjects and generate improvement recommendations,

    and to track learning progress over time.

    """

    def \_\_init\_\_(self, name: str, all\_scores: dict):

        """

        Initializes a new Student object.

        Args:

            name (str): The name of the student.

            all\_scores (dict): A dictionary where keys are academic periods (e.g., 'Year1', 'Year2')

                               and values are dictionaries of subject scores for that period.

                               Example: {'Year1': {'Math': 80, 'English': 75}, 'Year2': {'Math': 85, 'English': 80}}

        """

        super().\_\_init\_\_(name)

        self.all\_scores = all\_scores # Stores scores for all academic periods

        # For current calculations, we'll assume the 'current' scores are the latest ones available

        # This will need to be refined if specific year/term analysis is requested

        self.current\_period\_scores = self.\_get\_latest\_scores()

        self.average = self.calculate\_average(self.current\_period\_scores)

        self.gpa = self.calculate\_gpa(self.current\_period\_scores)

    def \_get\_latest\_scores(self) -> dict:

        """

        Helper method to get the scores from the latest academic period.

        Assumes periods are sortable (e.g., 'Year1', 'Year2').

        """

        if not self.all\_scores:

            return {}

        # Sort periods to get the latest one

        latest\_period = sorted(self.all\_scores.keys())[-1]

        return self.all\_scores[latest\_period]

    def calculate\_average(self, scores\_for\_period: dict) -> float:

        """

        Calculates the average score for a given set of scores (from a specific period).

        Args:

            scores\_for\_period (dict): A dictionary of subject scores for a specific period.

        Returns:

            float: The rounded average score. Returns 0.0 if no scores are present.

        """

        if not scores\_for\_period:

            return 0.0

        total = sum(scores\_for\_period.values())

        return round(total / len(scores\_for\_period), 2)

    def calculate\_gpa(self, scores\_for\_period: dict) -> float:

        """

        Calculates the Grade Point Average (GPA) for a given set of scores (from a specific period).

        Each score is converted to a grade point using the score\_to\_point method.

        Args:

            scores\_for\_period (dict): A dictionary of subject scores for a specific period.

        Returns:

            float: The rounded GPA. Returns 0.0 if no scores are present.

        """

        if not scores\_for\_period:

            return 0.0

        total\_points = sum(self.score\_to\_point(score) for score in scores\_for\_period.values())

        return round(total\_points / len(scores\_for\_period), 2)

    def score\_to\_point(self, score: float) -> float:

        """

        Converts a numerical score into a GPA point value based on a predefined scale.

        Args:

            score (float): The numerical score in a subject.

        Returns:

            float: The corresponding GPA point.

        """

        if score >= 70: return 5.0

        elif score >= 60: return 4.0

        elif score >= 50: return 3.0

        elif score >= 45: return 2.0

        elif score >= 40: return 1.0

        else: return 0.0

    def get\_weak\_subjects(self, period: str = None, threshold: float = 50.0) -> list[str]:

        """

        Identifies subjects where the student's score falls below a given threshold

        for a specific academic period. If no period is specified, uses the latest scores.

        Args:

            period (str, optional): The academic period (e.g., 'Year1'). If None, uses latest scores.

            threshold (float): The score threshold below which a subject is considered weak.

                               Defaults to 50.0.

        Returns:

            list[str]: A list of subject names where the student performed below the threshold.

                       Returns an empty list if no weak subjects or no scores for the period.

        """

        scores\_to\_check = self.all\_scores.get(period) if period else self.current\_period\_scores

        if not scores\_to\_check:

            return []

        weak\_subjects = []

        for subject, score in scores\_to\_check.items():

            if score < threshold:

                weak\_subjects.append(subject)

        return weak\_subjects

    def get\_improvement\_recommendations(self, period: str = None) -> str:

        """

        Generates general recommendations for improvement based on weak subjects

        for a specific academic period. If no period is specified, uses the latest scores.

        Args:

            period (str, optional): The academic period (e.g., 'Year1'). If None, uses latest scores.

        Returns:

            str: A formatted string containing recommendations, or a message

                 if no specific weaknesses are identified.

        """

        weak\_subjects = self.get\_weak\_subjects(period=period)

        if not weak\_subjects:

            return "Excellent performance! Keep up the great work across all subjects."

        else:

            recommendations = "Areas for improvement: " + ", ".join(weak\_subjects) + ".\n"

            recommendations += "Consider focusing extra study time or seeking additional help in these subjects."

            return recommendations

    def get\_performance\_over\_time(self, subject: str = None) -> dict:

        """

        Retrieves a student's performance (average or specific subject score)

        across all recorded academic periods.

        Args:

            subject (str, optional): If provided, returns scores for this specific subject

                                     across periods. If None, returns average scores per period.

        Returns:

            dict: A dictionary where keys are academic periods and values are

                  either the average score or the specific subject score for that period.

        """

        performance\_data = {}

        for period, scores in sorted(self.all\_scores.items()): # Sort by period for chronological order

            if subject:

                if subject in scores:

                    performance\_data[period] = scores[subject]

            else:

                performance\_data[period] = self.calculate\_average(scores)

        return performance\_data

    def display\_info(self):

        """

        Overrides the display\_info method from the Person class to provide

        student-specific information including GPA and average score for the latest period.

        Returns:

            str: A formatted string with student's name, GPA, and average score.

        """

        return f"Name: {self.name} | GPA (Latest): {self.gpa:.2f} | Average (Latest): {self.average:.2f}"

### 5.6.3 grade\_calculator.py

# grade\_calculator.py

import csv

import os

import statistics

from collections import defaultdict # Used for easier aggregation of student scores

from models import Student # Import the Student class from our models module

"""

This module contains the GradeCalculator class, responsible for loading student data

from CSV files, now supporting multiple academic years/periods per student.

It also includes a new method to manually add or update student records in memory,

and a method to save the current student data back to a CSV file.

It performs class-level calculations (like average score), and ranks students

based on their GPA for the latest period. It handles file operations and

integrates with the Student model for learning progress tracking.

"""

class GradeCalculator:

    """

    Manages the loading, calculation, and ranking of student data,

    now capable of handling scores across multiple academic periods and

    allowing manual addition/update of student records, and saving to file.

    It acts as a central point for aggregating and analyzing student performance.

    """

    def \_\_init\_\_(self, filepath: str = None, department\_name: str = "Unknown"): # Added department\_name

        """

        Initializes the GradeCalculator.

        Args:

            filepath (str, optional): The path to the CSV file containing student names,

                                      academic periods, and scores. Can be None if

                                      this calculator is primarily for manual data entry.

            department\_name (str): The name of the department/class this calculator represents.

                                   Used for saving data back to CSV.

        """

        self.filepath = filepath

        self.students = [] # A list to store Student objects

        # Internal mapping for quick lookup of students by name

        self.\_students\_by\_name = {}

        self.department\_name = department\_name # Store department name for saving

    def load\_students(self):

        """

        Loads student data from the specified CSV file, now supporting multiple

        academic periods per student. It aggregates scores for each student

        across different years/periods before creating Student objects.

        Includes robust error handling for file not found and invalid data within the CSV.

        Raises:

            FileNotFoundError: If the specified CSV file does not exist.

            ValueError: If a score cannot be converted to a float.

        """

        if not self.filepath:

            print("No filepath provided for loading students. Skipping file load.")

            return

        if not os.path.exists(self.filepath):

            raise FileNotFoundError(f"Error: File not found at '{self.filepath}'. "

                                    f"Please ensure the CSV file exists and the path is correct.")

        # Use a defaultdict to aggregate scores for each student across different years

        # Structure: {'Student Name': {'Year': {'Subject': score, ...}, 'Department': 'DeptName'}}

        students\_raw\_data = defaultdict(lambda: {'periods': defaultdict(dict), 'department': self.department\_name})

        invalid\_rows\_count = 0

        expected\_headers = ["Name", "Department", "Year"]

        try:

            with open(self.filepath, newline='', encoding='utf-8') as csvfile:

                reader = csv.DictReader(csvfile)

                # Validate essential headers

                if not all(header in reader.fieldnames for header in expected\_headers):

                    print(f"Warning: CSV file '{self.filepath}' is missing one or more required columns "

                          f"('Name', 'Department', 'Year'). Skipping file.")

                    return

                for row\_num, row in enumerate(reader, start=2):

                    name = row.get("Name")

                    year = row.get("Year")

                    department\_in\_csv = row.get("Department")

                    if not name or name.strip() == "":

                        print(f"Warning: Skipping row {row\_num} due to missing or empty student name: {row}")

                        invalid\_rows\_count += 1

                        continue

                    if not year or year.strip() == "":

                        print(f"Warning: Skipping row {row\_num} for student '{name}' due to missing or empty academic year.")

                        invalid\_rows\_count += 1

                        continue

                    # Update department name if present in CSV, otherwise use the one from init

                    if department\_in\_csv and department\_in\_csv.strip():

                        students\_raw\_data[name]['department'] = department\_in\_csv.strip()

                    current\_period\_scores = {}

                    has\_valid\_scores = False

                    for subject\_header, score\_str in row.items():

                        if subject\_header in expected\_headers:

                            continue

                        try:

                            score = float(score\_str)

                            current\_period\_scores[subject\_header] = score

                            has\_valid\_scores = True

                        except (ValueError, TypeError):

                            print(f"Warning: Invalid score '{score\_str}' for '{subject\_header}' for student '{name}' in year '{year}' (row {row\_num}). This subject's score will be ignored for this period.")

                    if not has\_valid\_scores:

                        print(f"Warning: Student '{name}' in year '{year}' (row {row\_num}) has no valid scores. This period's data will be skipped for this student.")

                        invalid\_rows\_count += 1

                        continue

                    students\_raw\_data[name]['periods'][year] = current\_period\_scores

            # Now, create/update Student objects from the aggregated data

            # We need to preserve existing students if this is a subsequent load

            # or if manual data was added.

            for name, data\_for\_student in students\_raw\_data.items():

                all\_scores\_for\_student = data\_for\_student['periods']

                if all\_scores\_for\_student: # Ensure there's at least one year of data

                    # If student already exists, update their all\_scores

                    if name in self.\_students\_by\_name:

                        existing\_student = self.\_students\_by\_name[name]

                        # Merge new periods with existing ones, new periods overwrite old ones

                        for year, scores in all\_scores\_for\_student.items():

                            existing\_student.all\_scores[year] = scores

                        # Re-calculate current period average/gpa based on potentially new latest scores

                        existing\_student.current\_period\_scores = existing\_student.\_get\_latest\_scores()

                        existing\_student.average = existing\_student.calculate\_average(existing\_student.current\_period\_scores)

                        existing\_student.gpa = existing\_student.calculate\_gpa(existing\_student.current\_period\_scores)

                    else:

                        # Create new student if not existing

                        student = Student(name, all\_scores\_for\_student)

                        self.students.append(student)

                        self.\_students\_by\_name[name] = student

                else:

                    print(f"Warning: No valid score data found for student '{name}'. Skipping.")

                    invalid\_rows\_count += 1

            if invalid\_rows\_count > 0:

                print(f"\nFinished loading. Encountered {invalid\_rows\_count} row(s)/student(s) with issues.")

        except Exception as e:

            print(f"An unexpected error occurred while reading the CSV file: {e}")

            raise

    def add\_student\_record(self, name: str, year: str, scores: dict):

        """

        Manually adds a new academic record for a student or updates an existing one.

        If the student does not exist, a new Student object is created.

        If the student exists, their scores for the specified year are updated/added.

        Args:

            name (str): The name of the student.

            year (str): The academic period (e.g., '100L', '200L').

            scores (dict): A dictionary of subject scores for the given year.

        """

        if not name or not year or not scores:

            print("Error: Student name, academic year, and scores cannot be empty.")

            return

        # Validate scores

        for subject, score in scores.items():

            if not isinstance(score, (int, float)):

                print(f"Error: Invalid score type for {subject}. Scores must be numbers. Record not added.")

                return

            if not (0 <= score <= 100):

                print(f"Error: Score for {subject} ({score}) must be between 0 and 100. Record not added.")

                return

        student\_found = self.\_students\_by\_name.get(name)

        if student\_found:

            # Student exists, update their scores for the specific year

            student\_found.all\_scores[year] = scores

            # Re-calculate current period average/gpa if this is the latest year

            student\_found.current\_period\_scores = student\_found.\_get\_latest\_scores()

            student\_found.average = student\_found.calculate\_average(student\_found.current\_period\_scores)

            student\_found.gpa = student\_found.calculate\_gpa(student\_found.current\_period\_scores)

            print(f"Updated record for '{name}' for year '{year}'.")

        else:

            # Student does not exist, create a new one

            all\_scores\_for\_new\_student = {year: scores}

            new\_student = Student(name, all\_scores\_for\_new\_student)

            self.students.append(new\_student)

            self.\_students\_by\_name[name] = new\_student

            print(f"Added new student '{name}' with data for year '{year}'.")

    def save\_students(self):

        """

        Saves the current in-memory student data for this class back to its

        associated CSV file. If no filepath is set, it prompts the user for one.

        This method reconstructs the flat CSV format from the nested student data.

        """

        if not self.filepath:

            print("No file path associated with this class. Please enter a filename to save to.")

            new\_filepath = input("Enter filename (e.g., my\_new\_class.csv): ").strip()

            if not new\_filepath:

                print("Save cancelled: No filename provided.")

                return

            if not new\_filepath.lower().endswith('.csv'):

                new\_filepath += '.csv'

            self.filepath = new\_filepath

            print(f"Saving to new file: '{self.filepath}'")

        if not self.students:

            print("No student data to save.")

            return

        # Collect all unique subject headers across all students and all periods

        all\_subjects = set()

        for student in self.students:

            for period\_scores in student.all\_scores.values():

                all\_subjects.update(period\_scores.keys())

        # Define the fixed headers

        fixed\_headers = ["Name", "Department", "Year"]

        # Sort subjects alphabetically for consistent header order

        dynamic\_headers = sorted(list(all\_subjects))

        # Combine all headers

        fieldnames = fixed\_headers + dynamic\_headers

        try:

            with open(self.filepath, 'w', newline='', encoding='utf-8') as csvfile:

                writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

                writer.writeheader()

                for student in self.students:

                    # Sort periods to ensure consistent order in CSV

                    for year in sorted(student.all\_scores.keys()):

                        row\_data = {

                            "Name": student.name,

                            "Department": self.department\_name, # Use the department name from GradeCalculator

                            "Year": year

                        }

                        # Add subject scores for the current year

                        for subject, score in student.all\_scores[year].items():

                            row\_data[subject] = score

                        writer.writerow(row\_data)

            print(f"Successfully saved data for '{self.department\_name}' to '{self.filepath}'.")

        except Exception as e:

            print(f"Error saving data to '{self.filepath}': {e}")

    def class\_average(self) -> float:

        """

        Calculates the average score for all students in the class, based on their

        latest academic period's average score.

        Returns:

            float: The rounded class average score. Returns 0.0 if no students are loaded.

        """

        if not self.students:

            return 0.0

        return round(statistics.mean([s.average for s in self.students]), 2)

    def rank\_students(self) -> list[Student]:

        """

        Ranks students based on their GPA from their latest academic period in descending order.

        Returns:

            list[Student]: A sorted list of Student objects, from highest to lowest GPA.

        """

        return sorted(self.students, key=lambda s: s.gpa, reverse=True)

    def grade\_distribution(self) -> dict:

        """

        Calculates the distribution of grades (A, B, C, D, E, F) across all subjects

        of all students, using scores from their latest academic period.

        Returns:

            dict: A dictionary where keys are grade letters and values are the

                  count of subjects that received that grade.

        """

        grade\_count = {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0}

        for student in self.students:

            for score in student.current\_period\_scores.values():

                if score >= 70:

                    grade\_count['A'] += 1

                elif score >= 60:

                    grade\_count['B'] += 1

                elif score >= 50:

                    grade\_count['C'] += 1

                elif score >= 45:

                    grade\_count['D'] += 1

                elif score >= 40:

                    grade\_count['E'] += 1

                else:

                    grade\_count['F'] += 1

        return grade\_count

### 5.6.4 report\_generator.py

# report\_generator.py

import statistics # Import statistics for potential use in comparison

from grade\_calculator import GradeCalculator

from models import Student # Import Student to use its methods

"""

This module defines the ReportGenerator class, which is responsible for

generating and displaying various performance reports based on the data

processed by the GradeCalculator. It leverages the OOP design by interacting

with Student and GradeCalculator objects.

Now includes the generation and display of recommendations for student improvement,

the ability to show learning progress over academic periods, and

a new static method for comparing multiple loaded classes.

"""

class ReportGenerator:

    """

    Generates and displays comprehensive reports on student performance,

    class averages, grade distributions, learning progress over time,

    and provides functionality for comparing multiple classes.

    """

    def \_\_init\_\_(self, calculator: GradeCalculator):

        """

        Initializes the ReportGenerator with an instance of GradeCalculator.

        Args:

            calculator (GradeCalculator): An instance of GradeCalculator

                                          containing loaded student data.

        """

        self.calculator = calculator

    def display\_summary(self):

        """

        Displays a summary of the class performance, including total students

        and the overall class average score. It also shows the grade distribution

        across all subjects for the latest academic period.

        """

        print("\n" + "=" \* 35)

        print("  CLASS PERFORMANCE SUMMARY")

        print("=" \* 35)

        total\_students = len(self.calculator.students)

        class\_avg\_score = self.calculator.class\_average()

        print(f"Total Students: {total\_students}")

        print(f"Class Average Score (Latest Period): {class\_avg\_score:.2f}\n") # Clarify "Latest Period"

        print("GRADE DISTRIBUTION (across all subjects for latest period):")

        distribution = self.calculator.grade\_distribution()

        # Ensure consistent order for display

        ordered\_grades = ['A', 'B', 'C', 'D', 'E', 'F']

        for grade in ordered\_grades:

            count = distribution.get(grade, 0) # Use .get() to handle cases where a grade might not be present

            print(f"  {grade}: {count} subject(s)")

        print("=" \* 35)

    def display\_top\_students(self, limit: int = 5):

        """

        Displays a list of top students ranked by their GPA from the latest

        academic period, along with personalized improvement recommendations.

        Args:

            limit (int): The maximum number of top students to display. Defaults to 5.

        """

        print(f"\n" + "=" \* 35)

        print(f"  TOP {limit} STUDENTS (by GPA - Latest Period)")

        print("=" \* 35)

        ranked\_students = self.calculator.rank\_students()[:limit]

        if not ranked\_students:

            print("No students available to rank.")

            print("=" \* 35)

            return

        for idx, student in enumerate(ranked\_students, 1):

            # Using the polymorphic display\_info method from the Student class

            print(f"{idx}. {student.display\_info()}")

            # Display improvement recommendations for each student

            recommendation = student.get\_improvement\_recommendations()

            print(f"   Recommendation: {recommendation}\n")

        print("=" \* 35)

    def display\_all\_students(self):

        """

        Displays records for all students, including their GPA, average score

        (both for the latest period), and personalized improvement recommendations.

        Students are displayed in ranked order (highest GPA first).

        """

        print("\n" + "=" \* 35)

        print("  ALL STUDENT RECORDS")

        print("=" \* 35)

        all\_students = self.calculator.rank\_students() # Get all students, already ranked

        if not all\_students:

            print("No student records found.")

            print("=" \* 35)

            return

        for student in all\_students:

            # Using the polymorphic display\_info method from the Student class

            print(student.display\_info())

            # Display improvement recommendations for each student

            recommendation = student.get\_improvement\_recommendations()

            print(f"  Recommendation: {recommendation}\n")

        print("=" \* 35)

    def display\_student\_learning\_progress(self, student\_name: str):

        """

        Displays the learning progress of a specific student over different academic periods.

        Shows average score progress and scores in individual subjects.

        Args:

            student\_name (str): The name of the student whose progress is to be displayed.

        """

        student\_found = None

        for student in self.calculator.students:

            if student.name.lower() == student\_name.lower():

                student\_found = student

                break

        if not student\_found:

            print(f"\nError: Student '{student\_name}' not found in the current class.")

            return

        print(f"\n" + "=" \* 35)

        print(f"  LEARNING PROGRESS FOR {student\_found.name.upper()}")

        print("=" \* 35)

        # Display overall average score progress

        avg\_progress = student\_found.get\_performance\_over\_time()

        if avg\_progress:

            print("Average Score Progress:")

            for period, avg\_score in avg\_progress.items():

                print(f"  {period}: {avg\_score:.2f}")

        else:

            print("  No average score data available for different periods.")

        print("\nSubject-wise Progress:")

        # Get all unique subjects across all periods for the student

        all\_subjects = set()

        for period\_scores in student\_found.all\_scores.values():

            all\_subjects.update(period\_scores.keys())

        if not all\_subjects:

            print("  No subject scores available for different periods.")

        else:

            for subject in sorted(list(all\_subjects)): # Sort subjects alphabetically

                subject\_progress = student\_found.get\_performance\_over\_time(subject=subject)

                if subject\_progress:

                    print(f"  {subject}:")

                    for period, score in subject\_progress.items():

                        print(f"    {period}: {score:.2f}")

                else:

                    print(f"  {subject}: No data for different periods.")

        print("=" \* 35)

    @staticmethod

    def compare\_classes(loaded\_classes: dict):

        """

        Compares the performance of multiple loaded classes/departments.

        Displays key metrics for each class side-by-side for easy comparison.

        Args:

            loaded\_classes (dict): A dictionary where keys are class names (str)

                                   and values are GradeCalculator instances.

        """

        print("\n" + "=" \* 50)

        print("  CLASS COMPARISON REPORT")

        print("=" \* 50)

        if len(loaded\_classes) < 2:

            print("Not enough classes loaded for comparison. Please load at least two.")

            print("=" \* 50)

            return

        # Prepare data for comparison

        comparison\_data = {}

        for class\_name, calculator\_instance in loaded\_classes.items():

            total\_students = len(calculator\_instance.students)

            class\_avg\_score = calculator\_instance.class\_average()

            # Get grade distribution for the latest period

            grade\_dist = calculator\_instance.grade\_distribution()

            comparison\_data[class\_name] = {

                "Total Students": total\_students,

                "Class Average Score (Latest)": class\_avg\_score,

                "Grade Distribution (Latest)": grade\_dist

            }

        # Print comparison header

        header = f"{'Metric':<30}"

        for class\_name in comparison\_data.keys():

            header += f"{class\_name:<20}"

        print(header)

        print("-" \* (30 + len(loaded\_classes) \* 20))

        # Print Total Students

        line = f"{'Total Students':<30}"

        for class\_name in comparison\_data.keys():

            line += f"{comparison\_data[class\_name]['Total Students']:<20}"

        print(line)

        # Print Class Average Score

        line = f"{'Class Average Score (Latest)':<30}"

        for class\_name in comparison\_data.keys():

            line += f"{comparison\_data[class\_name]['Class Average Score (Latest)']:<20.2f}"

        print(line)

        print("\nGrade Distribution (Latest Period):")

        ordered\_grades = ['A', 'B', 'C', 'D', 'E', 'F']

        for grade in ordered\_grades:

            line = f"  {grade} Count:"

            # Pad the grade count line to align with the metric column

            line = f"{line:<30}"

            for class\_name in comparison\_data.keys():

                count = comparison\_data[class\_name]['Grade Distribution (Latest)'].get(grade, 0)

                line += f"{count:<20}"

            print(line)

        print("\n" + "=" \* 50)

## 5.7 Successful Implementation with testing

 **Objective:** To ensure the reliability, correctness, and stability of the Student Performance Analysis CLI application's functionalities. Unit tests verify that individual components (classes and methods) operate as expected in isolation and integrate correctly.

 **Framework:** Python's built-in unittest module is utilized for test definition and execution.

 **Scope:** Comprehensive unit tests are implemented within a single test\_student.py module, covering all core functionalities across models.py, grade\_calculator.py, and report\_generator.py.

 **Key Aspects Tested:**

* **Data Models (Person, Student):** Initialization, GPA and average calculations, weak subject identification, improvement recommendation generation, and multi-period performance tracking.
* **Data Management (GradeCalculator):** CSV file loading (including multi-year data and error conditions), manual student record addition/update, and data saving to CSV.
* **Analytical Functions:** Class average, student ranking, and grade distribution calculations.
* **Reporting (ReportGenerator):** Display of various reports (summary, top students, all students, learning progress) and inter-class comparison.
* **Error Handling:** Validation of input and graceful handling of file-related and data-parsing errors.

 **Methodology:** Tests employ setUp and tearDown methods for managing test prerequisites and cleanup (e.g., temporary CSV files). unittest.mock.patch is used to simulate external interactions (like user input) and control side effects (like console printing) for isolated testing.

### 5.7.1. test\_student.py

import unittest

import os

import statistics

import csv

from unittest.mock import patch, mock\_open

from models import Student, Person # Import Person as well

from grade\_calculator import GradeCalculator

from report\_generator import ReportGenerator # Import ReportGenerator for class comparison test

class TestPerson(unittest.TestCase):

    """

    Unit tests for the base Person class.

    """

    def test\_person\_initialization(self):

        person = Person("John Doe")

        self.assertEqual(person.name, "John Doe")

    def test\_person\_display\_info(self):

        person = Person("Jane Smith")

        self.assertEqual(person.display\_info(), "Name: Jane Smith")

class TestStudent(unittest.TestCase):

    """

    Unit tests for the Student class, including multi-year data, GPA,

    average, weak subjects, and recommendations.

    """

    def setUp(self):

        # Example multi-year scores for a student

        self.multi\_year\_scores = {

            "100L": {"Math": 80, "English": 75, "Biology": 60},

            "200L": {"Math": 85, "English": 78, "Biology": 65},

            "300L": {"Math": 70, "English": 60, "Biology": 50}

        }

        self.student = Student("Test Student", self.multi\_year\_scores)

        # Student with no scores

        self.student\_no\_scores = Student("No Scores", {})

        # Student with some weak scores

        self.student\_weak = Student("Weak Student", {

            "100L": {"Math": 45, "English": 62, "Chemistry": 35}

        })

    def test\_gpa\_calculation\_latest\_period(self):

        # GPA for 300L: Math(70->5), English(60->4), Biology(50->3) = (5+4+3)/3 = 4.0

        expected\_gpa = round((5 + 4 + 3) / 3, 2)

        self.assertEqual(self.student.gpa, expected\_gpa)

    def test\_average\_score\_latest\_period(self):

        # Average for 300L: (70+60+50)/3 = 60.0

        expected\_average = round((70 + 60 + 50) / 3, 2)

        self.assertEqual(self.student.average, expected\_average)

    def test\_gpa\_calculation\_specific\_period(self):

        # GPA for 100L: Math(80->5), English(75->5), Biology(60->4) = (5+5+4)/3 = 4.67

        gpa\_100l = self.student.calculate\_gpa(self.multi\_year\_scores["100L"])

        self.assertEqual(gpa\_100l, round((5+5+4)/3, 2))

    def test\_average\_score\_specific\_period(self):

        # Average for 200L: (85+78+65)/3 = 76.0

        avg\_200l = self.student.calculate\_average(self.multi\_year\_scores["200L"])

        self.assertEqual(avg\_200l, round((85+78+65)/3, 2))

    def test\_student\_no\_scores(self):

        self.assertEqual(self.student\_no\_scores.average, 0.0)

        self.assertEqual(self.student\_no\_scores.gpa, 0.0)

        self.assertEqual(self.student\_no\_scores.get\_weak\_subjects(), [])

        self.assertIn("Excellent performance", self.student\_no\_scores.get\_improvement\_recommendations())

    def test\_get\_weak\_subjects(self):

        # Test with default threshold (50.0)

        weak\_subjects\_default = self.student\_weak.get\_weak\_subjects()

        self.assertIn("Math", weak\_subjects\_default)

        self.assertIn("Chemistry", weak\_subjects\_default)

        self.assertNotIn("English", weak\_subjects\_default) # English (62) is not below 50

        self.assertEqual(len(weak\_subjects\_default), 2)

    def test\_get\_weak\_subjects\_with\_custom\_threshold(self):

        # Test with custom threshold

        weak\_subjects\_custom = self.student\_weak.get\_weak\_subjects(threshold=60.0)

        self.assertIn("Math", weak\_subjects\_custom)

        # Corrected assertion: English (62) is NOT less than 60.0, so it should NOT be in weak subjects.

        self.assertNotIn("English", weak\_subjects\_custom)

        self.assertIn("Chemistry", weak\_subjects\_custom)

        self.assertEqual(len(weak\_subjects\_custom), 2) # Only Math and Chemistry are < 60

    def test\_get\_weak\_subjects\_specific\_period(self):

        # Test with specific period

        weak\_100l = self.student.get\_weak\_subjects(period="100L", threshold=78)

        self.assertIn("English", weak\_100l)

        self.assertIn("Biology", weak\_100l)

        self.assertNotIn("Math", weak\_100l)

    def test\_get\_improvement\_recommendations(self):

        recommendation\_strong = self.student.get\_improvement\_recommendations()

        self.assertIn("Excellent performance", recommendation\_strong)

        recommendation\_weak = self.student\_weak.get\_improvement\_recommendations()

        self.assertIn("Areas for improvement: Math, Chemistry", recommendation\_weak)

        self.assertIn("Consider focusing extra study time", recommendation\_weak)

    def test\_get\_performance\_over\_time\_average(self):

        expected\_avg\_progress = {

            "100L": round((80+75+60)/3, 2), # 71.67

            "200L": round((85+78+65)/3, 2), # 76.0

            "300L": round((70+60+50)/3, 2)  # 60.0

        }

        self.assertEqual(self.student.get\_performance\_over\_time(), expected\_avg\_progress)

    def test\_get\_performance\_over\_time\_subject(self):

        expected\_math\_progress = {

            "100L": 80,

            "200L": 85,

            "300L": 70

        }

        self.assertEqual(self.student.get\_performance\_over\_time(subject="Math"), expected\_math\_progress)

        self.assertEqual(self.student.get\_performance\_over\_time(subject="NonExistent"), {})

    def test\_student\_display\_info(self):

        # This will display info for the latest period (300L)

        expected\_gpa = round((5 + 4 + 3) / 3, 2)

        expected\_average = round((70 + 60 + 50) / 3, 2)

        expected\_info = f"Name: Test Student | GPA (Latest): {expected\_gpa:.2f} | Average (Latest): {expected\_average:.2f}"

        self.assertEqual(self.student.display\_info(), expected\_info)

class TestGradeCalculator(unittest.TestCase):

    """

    Unit tests for the GradeCalculator class, focusing on loading, adding,

    saving, and class-level calculations.

    """

    def setUp(self):

        self.test\_csv\_content = (

            "Name,Department,Year,Math,English,Biology\n"

            "John Doe,Science,100L,85,70,60\n"

            "John Doe,Science,200L,90,75,65\n"

            "Jane Smith,Arts,100L,70,80,55\n"

            "Jane Smith,Arts,200L,75,85,60\n"

        )

        self.temp\_filepath = "temp\_test\_students.csv"

        with open(self.temp\_filepath, 'w', newline='') as f:

            f.write(self.test\_csv\_content)

        self.calculator = GradeCalculator(self.temp\_filepath, department\_name="TestDept")

    def tearDown(self):

        if os.path.exists(self.temp\_filepath):

            os.remove(self.temp\_filepath)

    def test\_load\_students(self):

        self.calculator.load\_students()

        self.assertEqual(len(self.calculator.students), 2)

        john = self.calculator.\_students\_by\_name.get("John Doe")

        jane = self.calculator.\_students\_by\_name.get("Jane Smith")

        self.assertIsNotNone(john)

        self.assertIsNotNone(jane)

        self.assertEqual(john.name, "John Doe")

        self.assertEqual(len(john.all\_scores), 2) # 100L and 200L

        self.assertEqual(john.all\_scores["100L"]["Math"], 85)

        self.assertEqual(john.all\_scores["200L"]["English"], 75)

        self.assertEqual(john.gpa, round((5+5+4)/3, 2)) # John's 200L GPA

        self.assertEqual(jane.name, "Jane Smith")

        self.assertEqual(len(jane.all\_scores), 2) # 100L and 200L

        self.assertEqual(jane.all\_scores["100L"]["Biology"], 55)

        self.assertEqual(jane.all\_scores["200L"]["Math"], 75)

        self.assertEqual(jane.gpa, round((5+5+4)/3, 2)) # Jane's 200L GPA

    def test\_load\_students\_file\_not\_found(self):

        calc = GradeCalculator("non\_existent\_file.csv")

        with self.assertRaises(FileNotFoundError):

            calc.load\_students()

    @patch('builtins.print')

    def test\_load\_students\_no\_valid\_data(self, mock\_print):

        # Simulate a CSV with headers but no data rows, which won't print a warning from load\_students

        # Or, simulate a CSV with missing essential headers to trigger a warning from load\_students

        empty\_content\_missing\_header = "Name,Year,Math\n" # Missing 'Department' header

        with open(self.temp\_filepath, 'w', newline='') as f:

            f.write(empty\_content\_missing\_header)

        self.calculator.load\_students()

        self.assertEqual(len(self.calculator.students), 0)

        # Now, assert that the specific warning for missing headers was called

        mock\_print.assert\_called\_with(f"Warning: CSV file '{self.temp\_filepath}' is missing one or more required columns ('Name', 'Department', 'Year'). Skipping file.")

    def test\_add\_student\_record\_new\_student(self):

        self.calculator.load\_students() # Load initial data

        initial\_student\_count = len(self.calculator.students)

        self.calculator.add\_student\_record("New Student", "100L", {"Physics": 90, "Chemistry": 88})

        self.assertEqual(len(self.calculator.students), initial\_student\_count + 1)

        new\_student = self.calculator.\_students\_by\_name.get("New Student")

        self.assertIsNotNone(new\_student)

        self.assertEqual(new\_student.all\_scores["100L"]["Physics"], 90)

        self.assertEqual(new\_student.gpa, round((5+5)/2, 2))

    def test\_add\_student\_record\_update\_existing\_student\_new\_year(self):

        self.calculator.load\_students()

        self.calculator.add\_student\_record("John Doe", "300L", {"Math": 95, "English": 90, "Biology": 88})

        john = self.calculator.\_students\_by\_name.get("John Doe")

        self.assertEqual(len(john.all\_scores), 3) # Should now have 100L, 200L, 300L

        self.assertEqual(john.all\_scores["300L"]["Math"], 95)

        # Check if GPA/average updated to 300L scores

        self.assertEqual(john.gpa, round((5+5+5)/3, 2))

    def test\_add\_student\_record\_update\_existing\_student\_same\_year(self):

        self.calculator.load\_students()

        self.calculator.add\_student\_record("John Doe", "100L", {"Math": 99, "English": 98, "Physics": 97}) # Update 100L

        john = self.calculator.\_students\_by\_name.get("John Doe")

        self.assertEqual(len(john.all\_scores), 2) # Still 2 years, but 100L updated

        self.assertEqual(john.all\_scores["100L"]["Math"], 99)

        self.assertEqual(john.all\_scores["100L"]["Physics"], 97) # New subject added to 100L scores

    def test\_add\_student\_record\_invalid\_score\_type(self):

        self.calculator.load\_students()

        initial\_student\_count = len(self.calculator.students)

        with patch('builtins.print') as mock\_print:

            self.calculator.add\_student\_record("Bad Score Student", "100L", {"Math": "ninety"})

            self.assertEqual(len(self.calculator.students), initial\_student\_count) # No new student added

            mock\_print.assert\_called\_with("Error: Invalid score type for Math. Scores must be numbers. Record not added.")

    @patch('builtins.input', side\_effect=['new\_save\_file.csv'])

    def test\_save\_students\_new\_file(self, mock\_input):

        # Create a calculator with no initial filepath for this test

        calc\_manual = GradeCalculator(filepath=None, department\_name="ManualDept")

        calc\_manual.add\_student\_record("Manual Student", "100L", {"Art": 70.0, "Music": 80.0}) # Use floats for scores

        calc\_manual.add\_student\_record("Manual Student", "200L", {"Art": 75.0, "Music": 85.0})

        save\_filepath = "new\_save\_file.csv"

        if os.path.exists(save\_filepath):

            os.remove(save\_filepath)

        calc\_manual.save\_students()

        self.assertTrue(os.path.exists(save\_filepath))

        # Verify content

        with open(save\_filepath, 'r', newline='') as f:

            reader = csv.DictReader(f)

            headers = reader.fieldnames

            self.assertIn("Name", headers)

            self.assertIn("Department", headers)

            self.assertIn("Year", headers)

            self.assertIn("Art", headers)

            self.assertIn("Music", headers)

            rows = list(reader)

            self.assertEqual(len(rows), 2)

            self.assertEqual(rows[0]["Name"], "Manual Student")

            self.assertEqual(rows[0]["Year"], "100L")

            self.assertEqual(rows[0]["Art"], "70.0") # Expect "70.0" now

            self.assertEqual(rows[0]["Department"], "ManualDept")

            self.assertEqual(rows[1]["Name"], "Manual Student")

            self.assertEqual(rows[1]["Year"], "200L")

            self.assertEqual(rows[1]["Music"], "85.0") # Expect "85.0" now

        os.remove(save\_filepath) # Clean up

    def test\_save\_students\_existing\_file(self):

        self.calculator.load\_students() # Load initial data

        self.calculator.add\_student\_record("John Doe", "300L", {"Math": 95, "English": 90}) # Add new year

        self.calculator.add\_student\_record("New Guy", "100L", {"Science": 80}) # Add new student

        self.calculator.save\_students() # Save to temp\_test\_students.csv

        # Reload the file into a new calculator to verify

        reloaded\_calc = GradeCalculator(self.temp\_filepath, department\_name="ReTestDept")

        reloaded\_calc.load\_students()

        self.assertEqual(len(reloaded\_calc.students), 3) # John, Jane, New Guy

        john = reloaded\_calc.\_students\_by\_name.get("John Doe")

        new\_guy = reloaded\_calc.\_students\_by\_name.get("New Guy")

        self.assertIsNotNone(john)

        self.assertIsNotNone(new\_guy)

        self.assertEqual(len(john.all\_scores), 3) # Should have 100L, 200L, 300L

        self.assertEqual(john.all\_scores["300L"]["Math"], 95)

        self.assertEqual(new\_guy.all\_scores["100L"]["Science"], 80)

    def test\_class\_average(self):

        self.calculator.load\_students()

        # John (200L avg): (90+75+65)/3 = 76.666... -> 76.67

        # Jane (200L avg): (75+85+60)/3 = 73.333... -> 73.33

        # Class Avg = (76.67 + 73.33) / 2 = 75.0

        john\_latest\_avg = round((90+75+65)/3, 2)

        jane\_latest\_avg = round((75+85+60)/3, 2)

        expected\_class\_avg = round(statistics.mean([john\_latest\_avg, jane\_latest\_avg]), 2)

        self.assertEqual(self.calculator.class\_average(), expected\_class\_avg)

    def test\_rank\_students(self):

        self.calculator.load\_students()

        ranked = self.calculator.rank\_students()

        # John's 200L GPA: (5+5+4)/3 = 4.67

        # Jane's 200L GPA: (5+5+4)/3 = 4.67

        # In case of tie, original order from loading might be preserved or arbitrary.

        # We'll check if both are present and their GPAs are correct.

        self.assertEqual(len(ranked), 2)

        self.assertEqual(ranked[0].gpa, 4.67)

        self.assertEqual(ranked[1].gpa, 4.67)

        # Can't guarantee order if GPAs are identical, just check presence

        self.assertIn("John Doe", [s.name for s in ranked])

        self.assertIn("Jane Smith", [s.name for s in ranked])

    def test\_grade\_distribution(self):

        self.calculator.load\_students()

        # John (200L): Math 90(A), English 75(A), Biology 65(B) -> 2A, 1B

        # Jane (200L): Math 75(A), English 85(A), Biology 60(B) -> 2A, 1B

        # Total: 4A, 2B

        expected\_dist = {'A': 4, 'B': 2, 'C': 0, 'D': 0, 'E': 0, 'F': 0}

        self.assertEqual(self.calculator.grade\_distribution(), expected\_dist)

class TestReportGenerator(unittest.TestCase):

    """

    Unit tests for the ReportGenerator class, focusing on its display methods

    and the new class comparison feature.

    These tests primarily check if methods run without errors and print expected output.

    """

    def setUp(self):

        self.temp\_filepath\_physics = "temp\_physics.csv"

        self.temp\_filepath\_cs = "temp\_cs.csv"

        self.physics\_content = (

            "Name,Department,Year,Physics,Chemistry\n"

            "Alice,Physics,100L,80,75\n"

            "Alice,Physics,200L,85,80\n"

            "Bob,Physics,100L,50,45\n"

            "Bob,Physics,200L,55,50\n"

        )

        self.cs\_content = (

            "Name,Department,Year,Programming,Databases\n"

            "Charlie,CS,100L,90,85\n"

            "Charlie,CS,200L,92,88\n"

            "Diana,CS,100L,60,55\n"

            "Diana,CS,200L,65,60\n"

        )

        with open(self.temp\_filepath\_physics, 'w', newline='') as f:

            f.write(self.physics\_content)

        with open(self.temp\_filepath\_cs, 'w', newline='') as f:

            f.write(self.cs\_content)

        self.physics\_calc = GradeCalculator(self.temp\_filepath\_physics, department\_name="Physics")

        self.physics\_calc.load\_students()

        self.cs\_calc = GradeCalculator(self.temp\_filepath\_cs, department\_name="Computer Science")

        self.cs\_calc.load\_students()

        self.report\_physics = ReportGenerator(self.physics\_calc)

        self.report\_cs = ReportGenerator(self.cs\_calc)

    def tearDown(self):

        if os.path.exists(self.temp\_filepath\_physics):

            os.remove(self.temp\_filepath\_physics)

        if os.path.exists(self.temp\_filepath\_cs):

            os.remove(self.temp\_filepath\_cs)

    @patch('builtins.print')

    def test\_display\_summary(self, mock\_print):

        self.report\_physics.display\_summary()

        mock\_print.assert\_called() # Just check if print was called, content is complex to assert

    @patch('builtins.print')

    def test\_display\_top\_students(self, mock\_print):

        self.report\_physics.display\_top\_students

### 5.7.2. Output of the test

After successful implementation and running the tests, it came out as thus:

**Added new student 'Manual Student' with data for year '100L'.**

**Updated record for 'Manual Student' for year '200L'.**

**No file path associated with this class. Please enter a filename to save to.**

**Saving to new file: 'new\_save\_file.csv'**

**Successfully saved data for 'ManualDept' to 'new\_save\_file.csv'.**

**.................**

**----------------------------------------------------------------------**

**Ran 28 tests in 0.604s**

**OK**

### 5.7.3. Performance Tool Summary sample

**===================================**

**TOP 2 STUDENTS (by GPA - Latest Period)**

**===================================**

**1. Name: Akinpeju Oluwadarasimi | GPA (Latest): 5.00 | Average (Latest): 91.25**

**Recommendation: Excellent performance! Keep up the great work across all subjects.**

**2. Name: Olajide Zainab | GPA (Latest): 2.76 | Average (Latest): 59.06**

**Recommendation: Consider focusing extra study time or seeking additional help in these subjects!**

# 6. Conclusion

The Student Performance Analysis CLI application successfully delivers a robust, modular, and user-friendly tool for academic data analysis. By adhering to strong Object-Oriented Programming principles, implementing comprehensive error handling, and providing features for learning progress tracking, class comparison, and data persistence, the project meets all specified requirements. The application serves as a testament to the effective application of learned course modules in solving a meaningful real-world problem, providing valuable insights for improving student academic outcomes.