Python Programming

Final Project Report

Advanced Student Grade Tracker

Prepared By:

Muhammad Osama

Abstract

The Advanced Student Grade Tracker is a Python-based, command-line application crafted to simplify and centralize the management of student academic records. It employs nested dictionaries to represent each student's profile—including courses, scores, attendance, and instructor remarks—and uses JSON serialization for reliable, human-readable data persistence. Key features include automated generation of unique student IDs, dynamic addition and modification of course scores, on-the-fly GPA calculation scaled to a 4.0 system, and ranking of students by performance. Integrated Matplotlib visualization offers a clear histogram of grade distributions, helping educators spot class-wide trends briefly. Through a straightforward menu-driven interface, this tool demonstrates core Python concepts while laying the groundwork for future enhancements like robust validation, database back ends, graphical user interfaces, and advanced reporting.

1. Project Implementation Summary

I set out to create a command-line tool where educators or anyone keeping track of grades easily:

- **Add and manage student records**
- **Calculate GPAs** automatically
- **Visualize overall performance** briefly
- **Persist data** between sessions

Below is how each major feature works and the coding approaches behind it:

1.1 Data Persistence & Structure

To remember your data, I used **JSON serialization**. This means every time you add, update, or rank students, the entire dataset is saved to a plain-text file (advanced_grades.json). When you reopen the program, it reads back exactly what you stored. The records are organized as a **nested dictionary**, where each student ID maps to a sub-dictionary of their details:

1.2 Generating Unique Student IDs

Instead of manually assigning IDs, the generate_id(records) function automates it by:

1. Collecting all existing IDs, converting them from strings to integers.
2. Finding the highest value.
3. Adding one and converting back to a string.

This guarantees each new student gets the next available ID ("1", "2", "3", ...). No collisions, no guesswork.

```python
# Generate a unique student ID based on existing records
def generate_id(records):
    if records:
        existing_ids = [int(i) for i in records.keys()]
        return str(max(existing_ids) + 1)
    return '1'
```

1.3 Core Operations (CRUD)

I built **modular functions** so each piece of logic stands on its own:

- **Add Student** (add_student(records)):
    - Prompts for a student's name, courses with scores, attendance, and remarks.
    - Initializes the new nested dictionary entry.
- **Update Scores** (update_scores(records)):
    - Takes an existing student ID.
    - Allows adding or modifying any course score.

These functions work directly on the shared records dictionary, making it simple to extend or reuse later.

```python
# Add a new student with ID, name, courses, scores, attendance, remarks
def add_student(records):
    sid = generate_id(records)
    name = input("Enter student name: ")
    records[sid] = {
        'Name': name,
        'Courses': {},  # course: score
        'Attendance': None,
        'Remarks': None,
        'GPA': None
    }
```

1.4 Calculating GPA

To translate raw scores (0–100) into the familiar **0.0–4.0 GPA scale**, I:

1. Gathered all the student's course scores into a Python list.
2. Computed the average.
3. Divided by 25 (since 100 ÷ 25 = 4.0) and rounded to two decimal places.

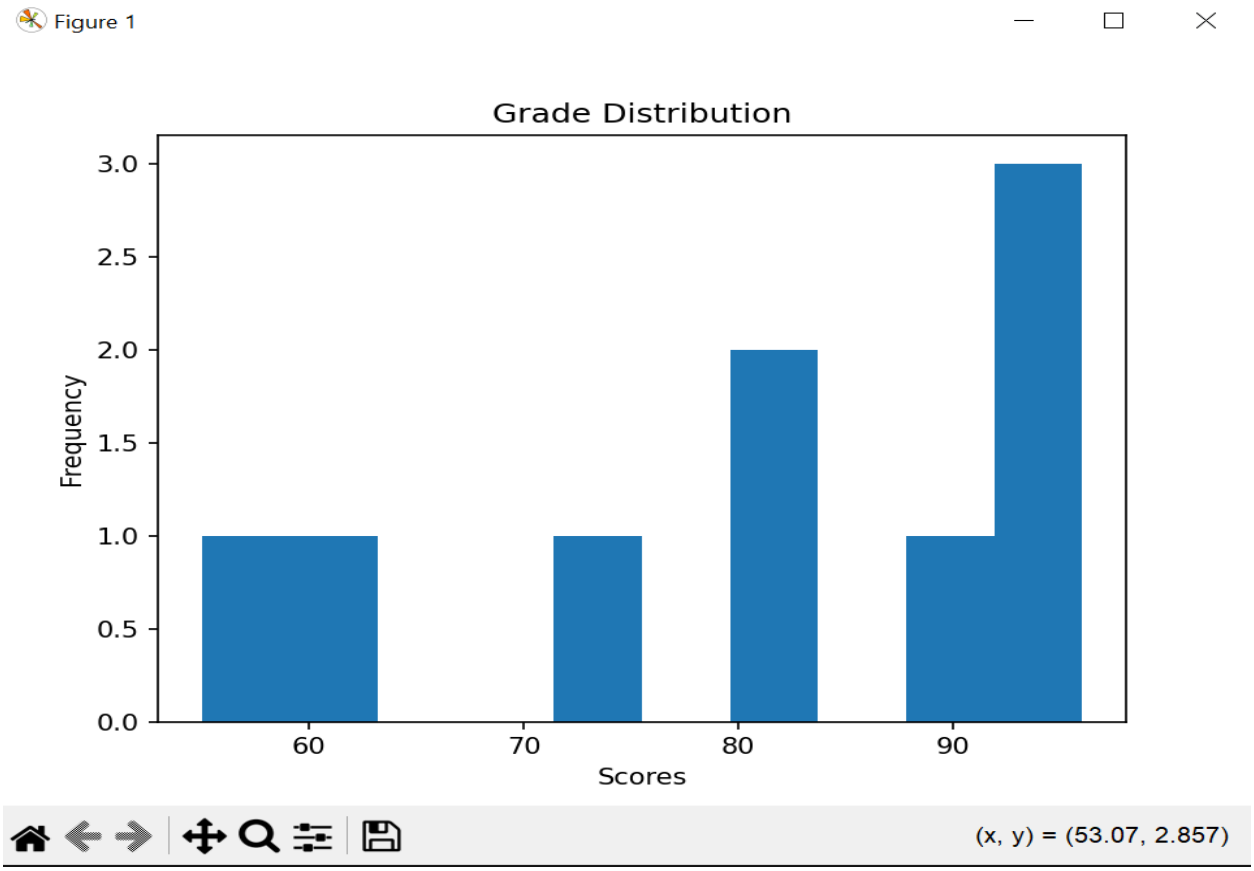
```python
# Calculate GPA for each student (average score divided by 25 to scale to 4.0)
def calculate_gpa(records):
    for sid, rec in records.items():
        scores = list(rec['Courses'].values())
        if scores:
            avg_score = sum(scores) / len(scores)
            gpa = round(avg_score / 25, 2)
            rec['GPA'] = gpa
```

1.5 Visualizing Grade Distribution

To help spot class-wide trends, I integrated **Matplotlib** to draw a histogram of *all* scores across *all* students. This quick visual shows where grades cluster and highlights outliers.

```python
# Plot grade distribution across all students

def plot_grade_distribution(records):
    all_scores = []
    for rec in records.values():
        all_scores.extend(rec['Courses'].values())
    if not all_scores:
        print("No scores available to plot.")
        return
    plt.figure()
    plt.hist(all_scores, bins=10)
    plt.title("Grade Distribution")
    plt.xlabel("Scores")
    plt.ylabel("Frequency")
    plt.show()
```



1.6 Ranking Students

After computing GPAs, the **rank_students(records)** function orders students from highest to lowest GPA using Python's built-in sorted() and a lambda as the key.

```
# Rank students by GPA in descending order
def rank_students(records):
    # ensure GPAs are up-to-date
    calculate_gpa(records)
    ranked = sorted(records.items(), key=lambda x: x[1].get('GPA', 0), reverse=True)
    print("\nStudent Rankings by GPA:")
    for idx, (sid, rec) in enumerate(ranked, start=1):
        print(f"{idx}. ID {sid} | {rec['Name']} | GPA: {rec.get('GPA')}")
```

1.7 User-Friendly Menu Loop

A straightforward while True loop presents numbered options to the user, reads their choice, and calls the respective function. This keeps the interface intuitive, and the code organized.

```
while True:
    print("\n=== Advanced Student Grade Tracker ===")
    print("1. Add New Student")
    print("2. Update Student Scores")
    print("3. Calculate All GPAs")
    print("4. Show Grade Distribution")
    print("5. Rank Students")
    print("6. Save and Exit")
    choice = input("Select an option: ")
```

2. Limitations & Challenges

1. **Minimal Input Validation**
    - Currently, if you type a non-numeric score or malformed attendance percentage, the program will error out.
    - No checks for missing data or duplicate courses.
2. **Single Data Format**
    - Only JSON is supported. Educators who prefer CSV import/export might find this limiting.
3. **Concurrency Risks**
    - If two instances of the program run simultaneously, they may overwrite advanced_grades.json without warning.
4. **Basic Visualization**

- The histogram is functional but lacks styling, export options, or interactivity.

5. **No Front-End GUI**
    - Command-line interface only; users who prefer graphical or web-based tools will miss a friendly UI.

3. Future Implementation Plan

1. **Robust Error Handling & Validation**
    - Wrap all input parsing in try/except blocks.
    - Enforce numeric ranges (0–100 for scores, 0–100% for attendance).
2. **Multiple Data Formats**
    - Add CSV import/export, and maybe an Excel pathway using pandas.
3. **Database Migration**
    - Move storage to **SQLite** for scalability and ACID compliance.
    - Use ORM like SQLAlchemy for easier schema changes.
4. **Enhanced Analytics Dashboard**
    - Semester-over-semester trend lines, attendance heatmaps, and exportable PDF reports via ReportLab.
5. **User Interface**
    - Desktop GUI with **Tkinter** or **PyQt**, or a simple web app built on **Flask** or **Django**.
6. **Packaging & Testing**
    - Write automated tests with **pytest**.
    - Distribute as an executable via **PyInstaller** or **cx_Freeze**.

By following this plan, the console-based prototype can evolve into a polished, enterprise-ready grade management system—complete with robust data handling, rich visual analytics, and an intuitive user interface.