Python Term Project – Student Grade Tracking System

1. Project Overview

- **Goal:** Create a terminal-based platform for instructors to manage student rosters, grade records, and performance analytics.
- **Scenario:** A college department needs a lightweight grading utility that instructors can run locally without relying on spreadsheets.
- **Key Competencies:** nested data structures, calculations, file persistence, modular architecture, and optional OOP.

2. Learning Outcomes

- Capture multi-course grading data using dictionaries or classes.
- Calculate averages, weighted scores, and letter grades.
- Persist rosters and grades across sessions with JSON or CSV.
- Automate analytics such as highest/lowest performers and grade distributions.
- Build user-friendly menus for instructors and teaching assistants.

3. Deliverables

1. `README.md` describing setup, grading workflows, and sample input.
2. Python modules:
    - `main.py` – CLI entry point and navigation.
    - `roster.py` – student and course management.
    - `grades.py` – grade entry, updates, and calculations.
    - `analytics.py` – summaries, distributions, and progress reports.
    - `storage.py` – data persistence, backups, schema validation.
3. Runtime data files (`data/students.json`, `data/courses.json`, `data/grades.json`).
4. Automated tests covering average calculations, grade updates, and validation rules.

4. Functional Requirements

4.1 Roster & Course Management

- Maintain student profiles (ID, name, email) and course rosters (code, title, term, instructor).
- Enroll/unenroll students in courses with history tracking.
- Optional: support sections or lab groups.

Required Functions (`roster.py`):

```
def load_students(path: str) -> list: ...
def save_students(path: str, students: list) -> None: ...
def add_student(students: list, student_data: dict) -> dict: ...
def update_student(students: list, student_id: str, updates: dict) ->
dict: ...
```

```
def enroll_student(course_roster: dict, student_id: str) -> dict: ...
def withdraw_student(course_roster: dict, student_id: str) -> dict: ...
```

4.2 Grade Recording

- Record assignments, projects, exams, or participation grades with weights.
- Support grade edits, deletions, and commentary (e.g., late penalty).
- Validate grades fall within configured range (0-100 by default).

Required Functions (`grades.py`):

```
def record_grade(gradebook: dict, course_id: str, student_id: str,
assessment: dict) -> dict: ...
def update_grade(gradebook: dict, course_id: str, student_id: str,
assessment_id: str, new_score: float) -> dict: ...
def delete_grade(gradebook: dict, course_id: str, student_id: str,
assessment_id: str) -> dict: ...
def calculate_student_average(gradebook: dict, course_id: str, student_id:
str) -> float: ...
def calculate_course_average(gradebook: dict, course_id: str) -> float:
...
```

4.3 Analytics & Reporting

- Generate grade distributions (histograms or letter-grade counts).
- Identify highest and lowest performers per course.
- Provide progress reports per student summarizing assessments completed, missing work, and current standing.
- Export printable reports to `reports/`.

Required Functions (`analytics.py`):

```
def grade_distribution(gradebook: dict, course_id: str, bins: list[int]) -
> dict: ...
def top_performers(gradebook: dict, course_id: str, limit: int = 5) ->
list: ...
def student_progress_report(gradebook: dict, course_id: str, student_id:
str) -> dict: ...
def export_report(report: dict, filename: str) -> str: ...
```

4.4 Grading Policies & Configuration

- Allow instructors to define grade categories with weights that sum to 100%.
- Support optional letter grade scales (A-F) with customizable thresholds.
- Handle incomplete grades and apply penalties for late submissions.

Required Functions (`grades.py` / `config.py`):

```
def set_grade_policy(course_settings: dict, course_id: str, policy: dict)
-> dict: ...
def compute_weighted_score(scores: list[dict], policy: dict) -> float: ...
def assign_letter_grade(score: float, scale: dict) -> str: ...
```

4.5 Data Persistence & Backups

- Separate storage for students, courses, gradebook entries, and configuration.
- Automatic backups upon major operations (e.g., after grade import).
- Validate schema compatibility before loading.

Required Functions (`storage.py`):

```
def load_state(base_dir: str) -> tuple[list, list, dict, dict]: ...
def save_state(base_dir: str, students: list, courses: list, gradebook:
dict, settings: dict) -> None: ...
def backup_state(base_dir: str, backup_dir: str) -> list[str]: ...
def import_from_csv(csv_path: str, course_id: str) -> dict: ...
```

5. User Experience Requirements

- Provide instructor dashboard summarizing in-progress courses.
- Guided prompts when entering grades (course → assessment → score).
- Show warnings for missing weights or unassigned categories.
- Confirm bulk imports or deletions.

6. Validation & Rules

- Prevent duplicate assessment IDs.
- Ensure weight totals equal 100% before computing overall averages.
- Block deletion of students who still have recorded grades unless archival flow is followed.
- Handle invalid menu choices gracefully.

7. Testing Expectations

- Unit tests for weighted score calculations, letter grade assignment, and enrollment validation.
- Include mock datasets for deterministic tests.

8. Suggested Timeline

1. Week 1 – Roster and course setup.
2. Week 2 – Grade recording and validation.
3. Week 3 – Analytics and reporting.
4. Week 4 – Configuration, backups, and UI refinements.
5. Week 5 – Testing, README, and final polish.

9. Grading Rubric (100 pts)

- Roster & Course Management: 20 pts
- Grade Recording & Calculations: 25 pts
- Policy & Configuration Handling: 15 pts
- Data Persistence & Backups: 15 pts
- Analytics & Reporting: 10 pts
- Documentation & UX: 10 pts
- Testing & Code Quality: 5 pts

10. Submission Checklist

- [ ] Sample grade policy and dataset provided.
- [ ] README walks through entering grades and generating reports.
- [ ] Tests included and passing.
- [ ] Weekly commits demonstrate incremental progress.