CSE 2050 Data structures and Object-Oriented Design — Spring 2026

Project Milestone #1

Hello everyone !! Welcome to Milestone #1 of the group project. I hope you have your project partner next to you and are excited to work together. This group project is designed to help you connect course concepts to real problem-solving, gain confidence working on larger programs, and strengthen teamwork and communication skills. This semester, you will complete a scaffolded project that highlights how selecting different and more efficient data structures can improve program design and performance.

In this project, you will be working with simulated student and course data and will find ways to efficiently store and perform operations on it.

Universities maintain extensive records related to students, courses, credits, enrollment, and grades. In a production environment, this information is managed through databases and institutional software. In this milestone of the project, you will apply a core Object-Oriented skill: Modeling Real-World entities using clear, well-structured classes.

You are given two CSV files university_data.csv and course_catalog.csv .
Here is a preview of these CSV files:

Student ID	**Name**	**Courses**
STU00001	Student_1	ENG1010:B-;MATH2010:C;CSE1010:A-;
STU00012	Student_12	BIO1010:D;PHYS1010:B;ECON1010:D

Table 1: Contents in university_data.csv

Course Code	**Credits**
CSE1010	2
CSE2050	3
CSE3100	2

Table 2: Contents in course_catalog.csv

Your task is to take the data in these two files and represent them as objects of elegantly defined classes.

Let us take a deeper look at how to layout this project and all the minimum requirements.

Project Tasks

General expectations

- Use clean object-oriented design: each class should manage its own data and responsibilities.
- Use appropriate Python data types.
- Your methods should be small, readable, and well documented.
- You don't have to use the method names given below, but you should implement the desired functionality presented below.

- Data will be loaded from CSV files, so your design should work for many inputs, class design should not depend upon the types of data file.

Task 1: Implement the `Course` Class

A `Course` represents a single course in the university catalog (e.g., `CSE2050`).

Required Fields

- `course_code` (*string*) — unique identifier for the course (e.g., `"CSE1010"`).
- `credits` (*integer*) — number of credits earned for completing the course.
- `students` (*list of Student objects*) — all students enrolled in the course.

Required Methods

- `add_student(student)` — adds a `Student` object to the course roster.
- `get_student_count()` — returns the number of students currently enrolled.

Design note: The course is responsible for maintaining its own roster. A student enrolling in a course should cause both the student and the course to reflect that relationship. The students list should mandatorily be an object of Student class.

Task 2: Implement the `Student` Class

A `Student` represents an individual student and the set of courses they have taken (with grades).

Required Fields

- `student_id` (*string*) — unique identifier for the student.
- `name` (*string*) — the student's name.
- `courses` (*dictionary*) — a dictionary of the courses a student has takes :

  `Course object : grade`

  where `grade` is a letter grade such as `"A"`, `"B+"`, etc.

Required Methods

- `enroll(course, grade)` — enrolls the student in a course with the given grade and updates the course roster.
- `update_grade(course, grade)` — modify the student grade for a particular course
- `calculate_gpa()` — computes and returns the GPA using all graded courses and their credits.
- `get_courses()` — returns a list of course objects taken by the student.
- `get_course_info()` — returns a structured summary of all enrollments, including course code, grade, and credits.

Implementation guidance:

- Use the following letter-grade to grade-point mapping when computing GPA:

  ```
  GRADE_POINTS = {
    'A' : 4.0,  'A-' : 3.7,
    'B+': 3.3,  'B'  : 3.0,  'B-' : 2.7,
    'C+': 2.3,  'C'  : 2.0,  'C-' : 1.7,
    'D' : 1.0,
    'F' : 0.0
  }
  ```

- GPA must be weighted by course credits:

$$\text{GPA} = \frac{\sum(\text{grade\_points} \times \text{credits})}{\sum \text{credits}}$$

Task 3: Implement the `University` Class

The `University` class serves as the central manager. It stores all students and courses and provides methods to query enrollment information efficiently.

Required Fields

- `students` (*dictionary*) — maps `student_id` → `Student object`.
- `courses` (*dictionary*) — maps `course_code` → `Course object`.

Required Methods

- `add_course(course_code, credits)` — if the course does not exist, create and store it; return the course object.
- `add_student(student_id, name)` — if the student does not exist, create and store them; return the student object.
- `get_student(student_id)` — returns the student object for that ID (or `None` if not found).
- `get_course(course_code)` — returns the course object for that code (or `None` if not found).
- `get_course_enrollment(course_code)` — returns the number of students enrolled in the given course.
- `get_students_in_course(course_code)` — returns a list of student objects enrolled in the given course.

Design note: The dictionaries in `University` are meant to make lookup fast and simple. Use them to avoid repeatedly scanning lists to find a student or course.

Loading data from CSV

- Read student data and course data from the CSV files and create necessary objects and store those objects in a University Object.
- In Student info, courses and grades are given as a course_code1:grade1;course_code2:grade2;

Documentation

- Every method and every class should be properly documented including the name of the team member who designed the functionality
- Test cases and method calls should also be documented

Validation and Error Handling

- Reject invalid IDs and empty names. IDs are 8 character length strings and always start with STU .
- Prevent duplicate entries.
- Validate letter grades before storing or using them in GPA calculations.
- Avoid division-by-zero.

Demonstration

Your code should be perform below queries:

- Get the list of students enrolled in a course.
- Print GPA of a student.
- Print all the courses and course info (grades and credits) for a student
- Calculate mean, mode and median for a course
- Calculate mean and median for the GPA of all students in the university
- Print common students in two different courses (Intersection)

Test Cases

Your code should contain below test cases:

Course Class:

- Test object creation for course class
- Test adding student objects to the course roster
- Test that code prevents duplicate object entries in student roster
- Test student count

Student Class:

- Test object creation for student class
- Test enrolling to course

- Test GPA calculation
- Test getting student courses

University Class:

- Test object creation for university class
- Test adding a course to university object
- Test duplicate course objects
- Test adding a student
- Test adding duplicate student
- Test getting student info
- Test getting non-existent student info
- Test getting course
- Test getting non-existent course

What to submit

- Python source file(s) containing the required classes and CSV loading logic
- Python file containing test cases
- A short Readme file explaining how to run the program and the tests
- Submit all these files to Gradescope
- You have the choice to give the files appropriate names

Optional Enhancements (Encouraged)

- Additional queries like implementation of your version of Dean's List, transcript-style output
- Cleaner architecture, (helper methods,modular file structure)
- As long as Minimum requirements are fulfilled, Optional enhancements are welcome.

Collaboration expectations

- You will have designated work time during lab (after lab activity) to make progress and ask the TA questions
- The TA will confrim that responsibilities are shared and that both partners are actively contributing (code should contain the name of the person who designed and coded the functionality)
- Additional work outside of the class is expected – plan accordingly
- Both partners should be prepared to explain the design and demonstrate the program's functionality

Helpful Python Documentation (for this project)

This project uses a small set of core Python features repeatedly. The links below (and the topics they cover) are enough to complete the assignment confidently.

1. Classes and Object-Oriented Programming

- **Classes (official tutorial):**
  https://docs.python.org/3/tutorial/classes.html
  - Defining classes with `class Name:`
  - The `__init__` constructor and `self`
  - Instance attributes (e.g., `self.students`)
  - Methods and calling methods on objects
  - Special method `__str__` for printing objects
- **Data model: `__str__` (string representation):**
  https://docs.python.org/3/reference/datamodel.html#object.__str__

2. Dictionaries (critical for `University.students` and `University.courses`)

- **Built-in Types — Dictionaries:**
  https://docs.python.org/3/library/stdtypes.html#mapping-types-dict
  - Creating dictionaries: {} and `dict()`
  - Checking membership: `if key in d:`
  - Lookup with default: `d.get(key)` and `d.get(key, default)`
  - Iteration: `for key, value in d.items():`
- **Dictionary methods:**
  https://docs.python.org/3/library/stdtypes.html#dict
  - `items()`, `keys()`, `values()`

3. Lists and Membership Tests (used for course rosters)

- **Built-in Types — Lists:**
  https://docs.python.org/3/library/stdtypes.html#list
  - Appending: `lst.append(x)`
  - Membership checks: `if x not in lst:`
  - Slicing for samples: `lst[:5]`

4. Working with CSV Files (`csv.DictReader`)

- **`csv` module (official docs):**
  https://docs.python.org/3/library/csv.html
  - Reading with `csv.DictReader(f)` (each row becomes a dictionary)
  - Accessing fields by header name: `row['course_code']`
  - Converting strings to integers: `int(row['credits'])`
- **Example usage of `DictReader`:**
  https://docs.python.org/3/library/csv.html#csv.DictReader

5. String Processing (parsing the `courses` field)

- **String methods (`split`):**
  https://docs.python.org/3/library/stdtypes.html#str.split
- **Formatted string literals (f-strings):**
  https://docs.python.org/3/reference/lexical_analysis.html#f-strings
  - Printing nicely: `f"{course_code}:  {grade}"`
  - Field width formatting: `f"{course_code:12}"`

6. Common Patterns Used in This Code

- **Safe lookup (avoid crashes):**
  Use `dict.get(key)` to return `None` if the key is missing.
- **Computing weighted GPA:**
  Use a running total for points and credits.

$$\text{GPA} = \frac{\sum(\text{grade\_points} \cdot \text{credits})}{\sum \text{credits}}$$

  Round to 2 decimals with `round(x, 2)`:
  https://docs.python.org/3/library/functions.html#round
- **Sorting dictionary keys for clean output:**
  https://docs.python.org/3/library/functions.html#sorted

7. Debugging and Testing Tips

- **Printing and inspecting data structures:**
  https://docs.python.org/3/library/pprint.html
  - Use `pprint` when dictionaries/lists get large.
- **Exceptions (understanding common errors):**
  https://docs.python.org/3/tutorial/errors.html
  - `KeyError`, `ValueError`, `TypeError`
  - How to interpret tracebacks

Recommendation: Start by implementing the three classes and testing them manually with a few objects *before* loading the CSV files. Once the classes behave correctly, file loading becomes straightforward.