Practice problems for CS 122- Advanced Python Programming
Part B - Practical Section (80 points)

Instructions (apply to all options):

- Place all Python modules and test files in the same directory; do not create any subfolders.
- User the files names specified in the questions.
- Provide clear docstrings for every class and function.
- Write unit tests in a single file named `test_<module>.py`
- **You will be running this code on jupyterhub which will need you to run it from terminal, practice that.**
- **The practice questions are large. In the final exam, tasks will be slightly simpler, but they will all involve similar split of concepts.**
- Though not mandatory, you can create a main.py file to test your own code by calling functions in it.

Problem 1: Book Inventory and Analysis System (80 pts)

Objective: You will create a system to manage a collection of books. Your system must store book entries in a CSV file, persist them to a SQLite database via an ORM, and generate analytical reports showing average ratings by genre.

Task 1: `book.py` (15 pts)

- **1:** In `book.py`, define a class named `Book` that accepts five pieces of information: title, author, publication year, genre, and rating.
- **2:** In the constructor method of the `Book` class:
  - Check that the title and author are non-empty strings.
  - Check that the year is between 1500 and the current calendar year.
  - Check that the rating is a floating-point number between 0.0 and 5.0.
  - If any of these checks fail, raise a `ValueError` with an explanatory message.
- **3:** Add an instance method called `__str__` that returns a human-readable string in this format: `<title> by <author> (<year>) - <genre> [Rating: <rating>]`
- **4:** Add an function named `to_csv_row` that produces a list containing exactly these five values, in order: title, author, year, genre, rating.

Task 2: `inventory.py` (15 pts)

- **1:** In `inventory.py`, implement a function called `add_book` that takes two arguments: a `Book` object and a list of `Book` objects. This function should insert the new book into the list without returning anything.
- **2:** Implement a second function called `search_by_year` that takes a search year and a list of `Book` objects. Return a new list containing every `Book` whose year is equal to the search year.
- **3:** Write a function named `save_to_csv` that takes a list of `Book` objects and a file path. Use Python's built-in `csv` module to write a header row and then one row per book, using each book's `to_csv_row` method.
- **4:** Write a function named `load_from_csv` that takes a file path, reads the CSV file, reconstructs a `Book` object for each row, and returns the full list.
- After running the script, produce a file `books.csv` with at least five entries. (a test file will be given in exam, for practice you can add items and create it).

Task 3: `utils.py` (10 pts)

- **1:** In `utils.py`, create a decorator named `log_calls`. When applied to any function, this decorator should open (or create) a file named `logs.txt` in the folder and append a line each time the function is called. Each line must include the function's name, the values of its arguments, and the value it returns.
- **2:** In the same module, write a decorator named `validate_year` that can be applied to any function accepting a parameter called `year`. Before calling the wrapped function, validate that `year` falls between 1500 and the current year; if not, raise a `ValueError` with a clear message.
- **3:** Also in `utils.py`, implement a generator function named `filter_by_genre` that accepts a list of `Book` objects and a genre string. The generator should yield each `Book` whose genre matches the given string, ignoring case.
- Apply both decorators to at least one functions in `inventory.py` and produce a `logs.txt` file demonstrating their operation.

Task 4: `database.py` (15 pts)

- **1:** Set up SQLAlchemy ORM by defining `Base = declarative_base()`. Then create a class called `BookModel` that inherits from `Base` and maps to a table named `books`. Define columns for `id` (primary key), `title`, `author`, `year`, `genre`, and `rating`.
- **2:** In `database.py`, write a function named `init_db` that accepts an optional database URL string (defaulting to `sqlite:///../data/books.db`). Inside this function, create the database engine, call `Base.metadata.create_all(engine)` to create tables, and return a new `Session` object bound to the engine.
- **3:** Add a function named `add_book_db` that accepts a `Book` object and a SQLAlchemy `Session`. This function should create a `BookModel` instance from the `Book` data, add it to the session, and commit the session.
- **4:** Add a function named `get_all_books` that accepts a SQLAlchemy `Session`. This function should query all rows from `BookModel`, convert each row into a `Book` object, and return a list of these `Book` objects.

- initialize the database and store all books from books.csv to `books.db`.

Task 5: `analysis.py` (20 pts)

- **1:** In `analysis.py`, load the book data. You may choose to read the CSV file using `pandas.read_csv("books.csv")` or to open a SQLAlchemy session and query all books via your ORM functions. (In exam I can ask you to use a particular one, so practice for both)
- **2:** Build a pandas `DataFrame` with exactly three columns: "title", "genre", and "rating".
- **3:** Group the DataFrame by the "genre" column and calculate the mean of the "rating" column for each group.
- **4:** Save the resulting average ratings to a report file named `genre_ratings.csv`, without including the DataFrame index.
- **5:** Determine which five genres have the highest average ratings. (The exam file will have lots of data, for practice you can assume a smller number of genres)
- **6:** Use the seaborn library to plot a bar chart of those five genres and their average ratings. Label the axes and give the chart a descriptive title. Save the plot as `genre_plot.png`.

Task 6: `test_inventory.py` (5 pts)

- **1:** Create a file named `test_inventory.py` in the same directory as your modules.
- **2:** Write pytest functions that verify:
    1. Constructing a `Book` with valid data succeeds, and with invalid data raises `ValueError`.
    2. Calling `add_book` adds a `Book` to a list as expected.
    3. Calling `search_by_title` returns correct matches and an empty list when there
- This test module must be able to be run independently from terminal.

Problem 2: Movie Recommendation Tracker (80 pts)

Objective: Implement a movie tracker that records movies, stores them in CSV and SQLite, provides recommendations, and visualizes rating distributions.

Task 1: `movie.py` (15 pts)

- **1:** Create `movie.py`.
- **2:** Define a class named `Movie` that takes: title, director, genre, year, and rating.
- **3:** In its constructor, validate:
    - `title`, `director`, and `genre` must be non-empty strings.
    - `year` must be between 1900 and current year.
    - `rating` must be a float between 0.0 and 10.0.
    - Raise `ValueError` if validation fails.

- **4:** Implement `to_dict` returning a dictionary with keys `title`, `director`, `genre`, `year`, `rating`.

Task 2: `tracker.py` (15 pts)

- **1:** Create `tracker.py`.
- **2:** Define a class `MovieTracker` with an internal list to store `Movie` instances.
- **3:** Implement a method `add_movie(movie)` that appends a `Movie` to the list.
- **4:** Implement `remove_movie(title, year)` that removes and returns `True` if a matching movie is found, `False` otherwise.
- **5:** Implement `recommend_by_genre(genre, min_rating)` that returns a list of movies whose genre matches (case-insensitive) and whose rating is at least `min_rating`.
- **6:** Use list comprehensions and lambdas for filtering and sorting.

Task 3: `utils.py` (10 pts)

- **1:** Create `utils.py`.
- **2:** Write a decorator `track_time` that logs execution time of any function to `movie_logs.txt`, including function name and elapsed seconds.
- **3:** Write a generator function `stream_suggestions(movies)` that yields one `Movie` object at a time from a list.

Task 4: `db.py` (15 pts)

- **1:** Create `db.py`.
- **2:** Define SQLAlchemy `Base` and a model class `MovieModel` mapped to a table `movies` with columns matching `Movie` attributes.
- **3:** Write `setup_db()` that creates `movies.db` in the current directory and returns a Session object.
- **4:** Write `save_movie(movie, session)` to insert a `MovieModel` record.
- **5:** Write `load_movies(session)` to query all records, convert each to a `Movie`, and return a list.

Task 5: `analysis.py` (20 pts)

- **1:** Create `analysis.py`.
- **2:** Load movie data into a pandas DataFrame, using either CSV reading or ORM query.
- **3:** Ensure DataFrame has columns `genre` and `rating`.
- **4:** Compute a rating distribution summary (e.g., counts or frequencies) and save it to `rating_dist.csv`.
- **5:** Plot a histogram of movie ratings with seaborn and save as `rating_hist.png`.

Task 6: `test_tracker.py` (5 pts)

- **1:** Create `test_tracker.py`.

- **2:** Write tests to confirm:
    1. Creating valid and invalid `Movie` instances behaves as expected.
    2. `add_movie` and `remove_movie` handle additions and deletions correctly.
    3. `recommend_by_genre` returns appropriate suggestions.
    4. Database functions `save_movie` and `load_movies` persist and retrieve data.

Problem 3: Personal Expense Tracker (80 pts)

Objective: Build an expense tracker that logs expenses to CSV and SQLite, summarizes monthly spending, and visualizes top categories.

Task 1: `expense.py` (15 pts)

- **1:** Create `expense.py`.
- **2:** Define a class `Expense` with attributes: amount, category, date (as `datetime.date`), and description.
- **3:** In the constructor, ensure `amount` is positive and `category` is a non-empty string; otherwise raise `ValueError`.
- **4:** Implement a method `to_csv_row` returning a list `[amount, category, date.isoformat(), description]`.

Task 2: `tracker.py` (15 pts)

- **1:** Create `tracker.py`.
- **2:** Define a class `ExpenseTracker` with an internal list for `Expense` objects.
- **3:** Implement `add_expense(expense)` to add an `Expense`.
- **4:** Implement `remove_expense(index)` to delete the expense at the given list index, returning `True` if successful, `False` if index is out of range.
- **5:** Implement `summary_by_category(month, year)` that returns a dictionary mapping each category to the total amount spent in that month and year.

Task 3: `utils.py` (10 pts)

- **1:** Create `utils.py`.
- **2:** Write a decorator `timed` that logs function name and execution time to `expense_logs.txt`.
- **3:** Write a generator `monthly_report(expenses, month, year)` that yields tuples `(category, total_amount)` for each category in the specified month.

Task 4: `db.py` (15 pts)

- **1:** Create `db.py`.
- **2:** Define SQLAlchemy `Base` and a model `ExpenseModel` for table `expenses` with columns for amount, category, date, and description.
- **3:** Write `init_db()` to create `expenses.db` and return a Session.
- **4:** Write `save_expense(expense, session)` to insert a record.
- **5:** Write `get_expenses(session)` to return a list of `Expense` objects from all records.

Task 5: `analysis.py` (20 pts)

- **1:** Create `analysis.py`.
- **2:** Load expense data into a pandas DataFrame via CSV or ORM.
- **3:** For a given month and year, compute total spending per category and save to `expense_summary.csv`.
- **4:** Plot a bar chart of the top five spending categories and save as `expense_plot.png`.

Task 6: `test_tracker.py` (5 pts)

- **1:** Create `test_tracker.py`.
- **2:** Write tests for:
    1. Valid and invalid `Expense` creation.
    2. `add_expense` and `remove_expense` logic.
    3. Correct monthly summary output.
    4. Database persistence and retrieval.

Problem 4: Student Grade Management System (80 pts)

Objective: Create a grade book that records student scores, calculates statistics, and identifies honor students.

Task 1: `student.py` (15 pts)

- **1:** Create `student.py`.
- **2:** Define a class `Student` with `name`, `student_id`, and `grades` (a dictionary mapping course names to numeric grades).
- **3:** In the constructor, verify `name` and `student_id` are non-empty strings and grades mapping is non-empty; raise `ValueError` for violations.
- **4:** Implement a method `average_grade` that computes and returns the mean of all course grades.

Task 2: `grades.py` (15 pts)

- **1:** Create `grades.py`.

- **2:** Write a function `add_grade(student, course, grade)` that adds or updates the grade for the given course in the student's `grades` dictionary.
- **3:** Write a function `get_class_average(students)` that computes the average grade across all students and courses.

Task 3: `utils.py` (10 pts)

- **1:** Create `utils.py`.
- **2:** Implement a decorator `validate_grade` that ensures any grade passed to decorated functions is between 0.0 and 100.0; otherwise, raise `ValueError`.
- **3:** Implement a generator `honors_students(students, threshold)` that yields each `Student` whose average grade exceeds the threshold value.

Task 4: `database.py` (15 pts)

- **1:** Create `database.py`.
- **2:** Set up SQLAlchemy `Base` and define two models, `StudentModel` and `GradeModel`, mapping to tables `students` and `grades` respectively, with a foreign key relationship.
- **3:** Write `init_db()` to create `grades.db` and return a Session.
- **4:** Write `insert_student(student, session)` to add a `StudentModel` and its associated `GradeModel` records.
- **5:** Write `fetch_students(session)` to retrieve all students and their grades, constructing `Student` objects.

Task 5: `analysis.py` (20 pts)

- **1:** Create `analysis.py`.
- **2:** Load student and grade data into a pandas DataFrame with columns `student_id`, `course`, and `grade`.
- **3:** Compute summary statistics (mean, median, standard deviation) for each course and save to `grade_stats.csv`.
- **4:** Generate a seaborn boxplot showing grade distributions by course and save as `grade_boxplot.png`.

Task 6: `test_grades.py` (5 pts)

- **1:** Create `test_grades.py`.
- **2:** Write tests covering:
  1. Valid and invalid `Student` creation.
  2. `add_grade` and `get_class_average` correctness.
  3. Decorator `validate_grade` raising errors as appropriate.
  4. `honors_students` generator output.
  5. Database insertion and retrieval fidelity.

Problem 5: Fitness Progress Logger (80 pts)

Objective: Develop a workout logger that records exercise sessions, persists them, and visualizes performance.

Task 1: `workout.py` (15 pts)

- **1:** Create `workout.py`.
- **2:** Define a class `Workout` that requires: session_type (string), duration_minutes (integer), calories_burned (integer), and date (`datetime.date`).
- **3:** In the constructor, ensure `duration_minutes` and `calories_burned` are positive integers and `session_type` is non-empty; otherwise raise `ValueError`.
- **4:** Implement `to_dict` that returns a dictionary with keys matching each attribute.

Task 2: `logger.py` (15 pts)

- **1:** Create `logger.py`.
- **2:** Define a class `WorkoutLogger` with an internal list to store `Workout` objects.
- **3:** Implement `log_workout(workout)` to append to the internal list.
- **4:** Implement `get_workouts(start_date, end_date)` to return a list of `Workout` objects whose date falls between `start_date` and `end_date`, inclusive.

Task 3: `utils.py` (10 pts)

- **1:** Create `utils.py`.
- **2:** Write a decorator `validate_workout` that ensures any `Workout` passed to decorated functions meets the constructor requirements; otherwise raise `ValueError`.
- **3:** Write a generator `weekly_summary(workouts, week_start_date)` that yields tuples `(week_start_date, total_calories, total_duration)` for each week covered by the workouts list.

Task 4: `db.py` (15 pts)

- **1:** Create `db.py`.
- **2:** Set up SQLAlchemy `Base` and define a `WorkoutModel` mapping to table `workouts` with columns for session_type, duration_minutes, calories_burned, and date.
- **3:** Write `setup_db()` to create `workouts.db` and return a Session.
- **4:** Write `add_workout(workout, session)` to insert a record.
- **5:** Write `list_workouts(session)` to retrieve all records and return a list of `Workout` objects.

Task 5: `analysis.py` (20 pts)

- **1:** Create `analysis.py`.
- **2:** Load workout data into a pandas DataFrame via ORM or CSV.

- **3:** Calculate total calories burned and total duration per session_type over the last four weeks; save results to `calorie_summary.csv`.
- **4:** Plot a seaborn line chart showing trend of total calories burned per week; save as `calorie_trend.png`.

Task 6: `test_logger.py` (5 pts)

- **1:** Create `test_logger.py`.
- **2:** Write tests verifying:
    1. Valid and invalid `Workout` initialization.
    2. `log_workout` and `get_workouts` behavior.
    3. Decorator `validate_workout` raising errors appropriately.
    4. Database insertion and retrieval via `add_workout` and `list_workouts`.
