Program Design & Implementation Report

Automated Student Assignment/Attendance Registration and Attendance Rate Statistics System

[We will put your name here]

2025-05-20

Student ID: [Your ID]

a. Topic Problem Goal

The goal of this project is to develop a Python command-line application that automates the process of tracking student assignment submissions or class attendance. The system reads a list of students and their current records from a text file. It then scans a designated directory structure containing multiple subfolders, where each subfolder represents a specific assignment, lab, or check-in period. Within these assignment folders (and their sub-subfolders), the system identifies student submission files based on a specified file extension. It extracts student IDs from the filenames, updates the corresponding student's record to mark the submission/attendance, calculates total submissions and an attendance/submission rate for each student, and finally, writes the updated records back to the student list file. The system also provides detailed logging, error reporting, and options to query student data and view the attendance analysis in a formatted table.

b. Solution Conditions

- **Input Data Source 1 (Student Roster):**
  - A plain text file (e.g., `namelist.txt`).
  - Columns are separated by a Tab character (`\t`).
  - The first line (and subsequent lines) must contain at least the Student ID and Student Name. For instance: `StudentID\tStudentFullName\tMark1\tMark2\t...\tMarkN`.
  - The number of initial mark columns (e.g., `Mark1` to `MarkN`) is dynamically determined from the first valid data line in this file.
  - The file will be updated by the program to include two additional columns at the end of each student's record: `TotalSubmissions` and `SubmissionRate`.
- **Input Data Source 2 (Submissions Directory):**
  - A root directory (e.g., `./exercises/` or `./submissions/`) containing multiple direct subfolders.
  - Each direct subfolder within this root directory represents a unique assignment, lab, or check-in session (e.g., `./exercises/2025-03-03_01.04.57/`). The system processes these assignment folders in alphabetical order.

- **Input File Format (Student Submissions):**
  - Student submitted files are located within the aforementioned assignment subfolders. These files may be directly in the assignment folder (e.g., `ASSIGNMENT_FOLDER/student_file.py`) or nested within further sub-subfolders created by students (e.g., `ASSIGNMENT_FOLDER/STUDENT_CREATED_FOLDER/student_file.py`).
  - Filenames of submitted files **must contain the student's 8-digit ID** for the system to recognize and process them (e.g., `20240135_assignment1.py` or `C607-46_20241314.Wendy_Angelina.zip`).
  - The system searches for files of a specific type, identified by a file extension (e.g., `.py`, `.zip`, `.txt`), which is configurable via a command-line argument.
- **Output:**
  - **Updated Student List File:** An updated version of the input student list text file, with existing mark columns updated (a '1' indicates submission, '0' otherwise for each processed assignment), and two appended columns: the total number of submissions/attendances recorded for the student, and their submission/attendance rate (as a float formatted to two decimal places).
  - **Console Output Logs:** Detailed logs during program execution, including initialization messages, configuration, status of data loading, discovery of assignment folders, processing details for each folder (files found, IDs extracted, errors), and summaries.
  - **Processing Statistics for Each Check-in Sequence (Subfolder):** Includes files found, students marked, and error counts for that specific folder.
  - **Overall Processing Statistics:** Total files processed, total successful marks, total errors, and a list of all error files with reasons.
  - **Query Results (Optional):** Formatted display of a specific student's details when using the `query` command.
  - **Table View (Optional):** A formatted table displaying all students' data when using the `view` command or the `-view` option with `process`.
- **Constraints:**
  - Student ID is an 8-digit number (e.g., `20240135`).
  - Filenames of student submissions must contain this 8-digit student ID.
  - The student list file format must be tab-delimited. Initial structure: ID, Name, and initial mark columns.
  - The number of assignments processed is limited by the mark columns in `namelist.txt`.

c. Functional Requirements

- **Data Input & Loading:**
  - Load student data from tab-delimited text file into memory (list of dictionaries).
  - Dynamically determine assignment mark columns from the first valid line.
  - Handle pre-existing "Total" and "Rate" columns from previous runs.
  - Robust error handling for file issues and malformed lines.

- **File System Traversal & Identification:**
  - Identify and traverse direct subfolders in the root submissions directory (alphabetical order for assignments).
  - Recursively find submission files within each assignment subfolder matching a specified extension.
- **Information Extraction:**
  - Extract 8-digit student ID from submission filenames using regular expressions.
- **Data Processing & Updating:**
  - Look up student records by extracted ID.
  - Determine correct assignment column based on folder order.
  - Mark submission by setting the value to '1'.
  - Handle exceptions: ID not extractable, ID not found, assignment index out of bounds.
- **Result Calculation:**
  - Calculate total submissions for each student.
  - Calculate submission/attendance rate (Total Submissions / Assignments Considered).
- **Data Output & Reporting:**
  - Write updated records (marks, total, rate) back to the student list file, overwriting it.
  - Output detailed processing logs.
  - Output summary for each assignment folder and an overall processing summary.
  - List all error files with reasons.
- **CLI Functionality:**
  - Provide actions: `process`, `query`, `view`.
  - Allow CLI arguments for namelist, submissions directory, and file extension.
  - Use `tabulate` for formatted table outputs.

d. Code Style and Norms Requirements

The project implementation adheres to the following norms:

- **Follow PEP 8 Style Guide:** Including 4-space indentation, line length considerations, snake_case for variables/functions/modules, UPPER_CASE for constants, and appropriate blank lines.
- **Clear Naming:** Descriptive names for variables and functions.
- **Necessary Comments:** Docstrings for modules, classes, functions, and in-line comments for clarity.
- **Reasonable Function Division (Modularity):** Code organized into modules (`main.py`, `file_operations` `processing.py`, `reporting.py`, `config.py`) with distinct responsibilities.
- **Error Handling:** Use of `try-except` blocks, informative error/warning messages, and graceful handling of problematic conditions.

e. Rough Steps of the Solution

1. **Configure Parameters & Setup (CLI Parsing in `main.py`):** Parse arguments, initialize `Reporter`. Constants in `config.py`.

2. **Load Student Data (`file_operations.load_student_data`):** Open namelist, read lines, split by Tab, determine mark columns, handle existing Total/Rate, store as list of dictionaries. Error handling for malformed data.

3. **Identify Check-in (Assignment) Folders (`file_operations.discover_assignment_folders`):** Get direct subfolders in submissions root, sort alphabetically. Warn if folder count exceeds namelist capacity.

4. **Loop Through Check-in Folders & Process Submissions (in `main.py`'s `handle_process_action`):** For each assignment folder (up to namelist capacity):

   - Initialize folder stats.
   - Recursively get submission files (`file_operations.get_submission_files_in_folder`).
   - For each submission file:
     - Extract Student ID (`processing.extract_student_id`).
     - If ID extracted: Find student. If found & index valid: Mark submission (`processing.mark_submis`
       Else: Log error.
     - Else (ID not extracted): Log error.
   - Report Folder Summary (`reporter.folder_summary`).

5. **Calculate and Store Attendance/Submission Rate (`processing.calculate_final_statistics`):** For each student, calculate total and rate.

6. **Save Updated Data (`file_operations.save_student_data`):** Overwrite namelist with updated records.

7. **Output Overall Report (`reporter.overall_summary`):** Print overall stats and error list.

f. Program Flowchart

A visual flowchart would be included here, depicting the main program logic for actions like ’process’, ’query’, and ’view’, including decision points and loops as described in the textual report. (Refer to the textual description in section ’e’ for a step-by-step logical flow.)

Better still, if you have the flowchart, send it so that we insert it here.

g. Specific Code Implementation and Result

This section includes key code snippets and example program outputs.

Key Code Snippet: Student ID Extraction

From `attendance_processor/processing.py`:

```python
# processing.py
import re
from .config import STUDENT_ID_REGEX # STUDENT_ID_REGEX = r'(?<!\d)(\d{8})(?!\d)'
from typing import Optional # Ensure Optional is imported if not already
from .reporting import Reporter # Ensure Reporter is imported if not already

def extract_student_id(filename: str, reporter: Reporter) -> Optional[str]:
    """
    Extracts an 8-digit student ID from a filename using regex.
    """
    match = re.search(STUDENT_ID_REGEX, filename)
    if match:
        return match.group(1)
    return None
```

Listing 1: Student ID Extraction Logic

Example Console Output (Illustrative)

The following demonstrates typical console output during the 'process' action:

```
[INFO] Starting Student Submission Processor CLI...
[INFO] Action: Process Submissions
[INFO] Namelist file: namelist.txt
[INFO] Submissions directory: exercises
[INFO] Processing file extension: .py
[INFO] Namelist structure: Expecting 9 assignment mark column(s).
[INFO] Found 8 assignment folder(s): [2025-03-03_01.04.57, ..., 2025-04-21_01.40.00]
---
[INFO] Processing folder: '2025-03-03_01.04.57' (Assignment 1)
[INFO] Searching for '.py' files in and under 'exercises/2025-03-03_01.04.57'...
[INFO]   Found 23 '.py' file(s) in/under 'exercises/2025-03-03_01.04.57'.
[WARNING] File 'C607-06_hello world.py' in folder '2025-03-03_01.04.57': Could not extract student ID.
[WARNING] File 'C607-07_20241162_Tebenkov.py' in folder '2025-03-03_01.04.57': Student ID '20241162' not found in namelist.
...
==============================
[INFO] Summary for folder: 2025-03-03_01.04.57
  Files found (.py): 23
  Students successfully marked: 17
  File errors in this folder: 5
==============================
...
[INFO] Calculating totals and rates based on 8 assignment(s).
[INFO] Successfully saved updated student data to namelist.txt
--- Overall Processing Summary ---
[INFO] Overall Processing Summary
  Total assignment folders processed: 8
  Total submission files found: 256
  Total submissions successfully recorded: [Actual count]
  Total file-related errors encountered: 19
  Error File Details:
    - exercises/2025-03-03_01.04.57/C607-06/C607-06_hello world.py: Could not extract student ID.
    ... (list all other errors) ...
[INFO] Processing action complete.
```

Listing 2: Illustrative Console Output

h. Problems Encountered and Solutions

1. **Text File Parsing Issues:**

   - **Problem:** Namelist files might contain blank lines, inconsistent spacing, or malformed lines. Previously processed files would also contain "Total" and "Rate" columns, causing misinterpretation.

- **Solution:** Implemented checks to skip blank lines. Enhanced `load_student_data` to split lines by Tab, perform length checks, use `try-except` for mark parsing, and dynamically detect/handle pre-existing "Total"/"Rate" columns. Warnings are logged for malformed lines.

2. **Regular Expression Student ID Extraction Issue:**
   - **Problem:** Varied filename formats. Simple regex `\d{8}` could mismatch. Initial `\b(\d{8})\b` failed with underscores.
   - **Solution:** Refined regex to `(?<!\d)(\d{8})(?!\d)` for robust standalone 8-digit ID extraction, using negative lookbehind and lookahead.

3. **Recursive File Discovery in Assignment Folders:**
   - **Problem:** Initial design assumed files were directly in assignment folders. Needed to support student-created subfolders.
   - **Solution:** Modified `get_submission_files_in_folder` to use `os.walk()` for recursive traversal.

4. **Dynamic File Extension in Reporting:**
   - **Problem:** Log messages for file searches had hardcoded extensions.
   - **Solution:** Updated `Reporter.folder_summary` to accept `file_extension` as an argument, passed from `main.py` (`args.ext`).

5. **Handling More Assignment Folders than Namelist Mark Columns:**
   - **Problem:** Potential index errors if more assignment folders exist than available mark columns in the namelist.
   - **Solution:** The program determines available mark columns and processes assignments only up to this limit. A warning is logged if more folders are found. Rate calculation also respects this limit.

i. Project Learning Experience

Through completing this Final Project, I gained a deeper understanding of Python's capabilities in automating office tasks and data processing. I learned and practiced the following key skills:

- **File Operations and File System Interaction:** Reading/writing text files, parsing structured data, directory traversal (`os.listdir`, `os.walk`), and path manipulation (`os.path.join`).
- **String Processing and Regular Expressions:** String methods, crafting and applying robust regular expressions (`re` module) for ID extraction using lookarounds.
- **Data Structures and Manipulation:** Effective use of lists and dictionaries for managing student data.
- **Modular Program Design & Separation of Concerns:** Organizing code into logical modules for better readability and maintainability.
- **Command-Line Interface (CLI) Development:** Using `argparse` for user-friendly CLI with commands, arguments, and help messages.
- **Error Handling and Robustness:** Implementing `try-except` blocks, providing informative error messages, and handling edge cases.

- **Code Readability and Maintainability (PEP 8):** Adhering to styling guidelines and writing clear documentation.
- **Using External Libraries:** Integrating `tabulate` for enhanced console table presentation.

In summary, this project provided practical experience in developing a complete Python application, reinforcing the importance of planning, modular design, robust error handling, and clear user interaction. The iterative refinement of features was a valuable learning experience.