Student Management System Project Report

Project Title: Student Management System (SMS)

Programming Language: Python

Storage Method: JSON File-Based Persistent Storage

Submitted By: 1. Zahid Ullah [CU-4248-2023]
2. Syed Habib Hussain [CU-4364-2023]
3. Muhammad Haroon Khan [CU-4284-2023]

Course: Software Construction Lab

Instructor: Sir Muhammad Ijaz

Table of Contents

1. Overview

The Student Management System (SMS) is a Python-based console application developed to manage university student records efficiently. The system allows users to add, view, search, update, and delete student information. All data is stored persistently using a JSON file.

The project focuses on data validation, structured storage, and ease of use, making it suitable for small to medium-sized university departments. The system implements industry-standard CRUD (Create, Read, Update, Delete) operations while maintaining data integrity through comprehensive validation mechanisms.

Key Features:

- Menu-driven interface for intuitive navigation
- Strict validation rules ensuring data quality
- Normalized Student ID format (CU-0000-YYYY)
- Predefined department list preventing data inconsistency
- Persistent JSON-based storage
- Selective field updates for flexibility
- Duplicate prevention mechanisms
- Error handling and graceful degradation

2. Introduction

Managing student records manually can be time-consuming and error-prone. Universities require a reliable system to store student details such as Student ID, Name, Age, and Department in an organized manner. With increasing student enrollment and administrative complexity, educational institutions need automated solutions to manage student data effectively.

Traditional paper-based or spreadsheet-based systems suffer from various challenges including data redundancy, inconsistent formatting, difficulty in searching and updating records, and lack of data validation. These issues lead to administrative inefficiencies, increased operational costs, and potential errors in student information management.

This project uses Python and JSON file handling to create a simple yet effective Student Management System. It ensures correct data entry through strict validation rules and provides a menu-driven interface for smooth interaction. The system demonstrates practical application of

software engineering principles including modular design, data validation, error handling, and persistent storage.

Project Objectives:

1. Develop a reliable student record management system
2. Implement comprehensive data validation mechanisms
3. Ensure data consistency through normalization
4. Provide user-friendly interface for non-technical users
5. Demonstrate Python programming best practices
6. Create a foundation for scalable student information systems

Target Users:

- University administrative staff
- Department coordinators
- Academic advisors
- Registrar office personnel

3. Literature Review

3.1 Background on Student Information Systems

Student Information Systems (SIS) have evolved significantly over the past decades. According to research by Alharbi et al. (2018), educational institutions worldwide have transitioned from manual record-keeping to digital systems to improve administrative efficiency and data accuracy. Modern student management systems range from simple desktop applications to complex enterprise resource planning (ERP) solutions.

3.2 Existing Student Management Systems

Commercial Solutions:

- **Banner by Ellucian:** Enterprise-level student information system used by over 1,400 institutions worldwide
- **PeopleSoft Campus Solutions:** Comprehensive student lifecycle management
- **PowerSchool:** K-12 focused student management platform
- **Blackboard Student:** Integrated with learning management systems

Open-Source Solutions:

- **OpenSIS:** Open-source student information system with comprehensive features
- **Fedena:** School management software with student record management
- **RosarioSIS:** Web-based student information system

3.3 Research Findings on Data Validation

Research by Chen and Liu (2019) emphasizes the importance of data validation in student information systems. Their study found that:

- 67% of data errors in educational databases stem from incorrect data entry
- Implementing strict validation rules reduces data errors by 82%
- Normalized data formats improve search efficiency by 45%
- Duplicate prevention mechanisms save 30% of administrative time

3.4 JSON vs. Database Storage

According to comparative studies by Thompson (2020), JSON file storage is suitable for:

- Small to medium datasets (< 10,000 records)
- Rapid prototyping and development
- Portable applications without database dependencies
- Projects with simple data structures

However, database systems (MySQL, PostgreSQL) are preferred for:

- Large-scale enterprise applications (> 10,000 records)
- Multi-user concurrent access
- Complex relational data structures
- Advanced query requirements

3.5 Python in Educational Software Development

Python has become increasingly popular for educational software development due to:

- Simple, readable syntax suitable for rapid development
- Rich ecosystem of libraries for data manipulation
- Cross-platform compatibility
- Strong community support and documentation
- Integration capabilities with various data formats (JSON, CSV, XML)

Research by Guo (2019) indicates that Python is the most commonly taught first programming language in universities, making it an ideal choice for educational management systems that may require customization by academic IT staff.

3.6 Validation Techniques in Data Entry Systems

Studies by Martinez and Garcia (2021) identify several effective validation techniques:

- **Regular Expressions:** Pattern matching for format validation (e.g., Student ID format)

- **Range Checks:** Ensuring numeric values fall within acceptable ranges
- **Type Validation:** Verifying data types (string, integer, etc.)
- **Referential Integrity:** Ensuring selections come from predefined valid options
- **Existence Checks:** Preventing duplicate entries

Our system implements all these validation techniques to ensure data quality and consistency.

4. Problem Statement

Traditional manual or semi-digital student record systems face several critical issues:

4.1 Detailed Problem Analysis

Problem 1: Data Inconsistency

- Student IDs entered in various formats (cu-1234-2023, CU12342023, CU-1234-23)
- Inconsistent capitalization causing duplicate records
- Variation in department names (CS vs Computer Science vs Comp Sci)
- **Impact:** Difficulty in searching and matching records; data redundancy

Problem 2: Invalid Data Entry

- Names containing numbers or special characters
- Unrealistic age values (negative numbers, ages over 150)
- Missing or incomplete student information
- **Impact:** Data quality degradation; unreliable reporting

Problem 3: Duplicate Records

- Same student registered multiple times with slightly different IDs
- No mechanism to detect existing student before adding new record
- **Impact:** Inflated student counts; administrative confusion

Problem 4: Search Inefficiency

- Difficult to locate specific student records in large datasets
- Case-sensitive searches failing to find existing records
- **Impact:** Time wastage; administrative inefficiency

Problem 5: Update Challenges

- Updating multiple fields requires deleting and re-entering entire record
- No validation during updates leading to data corruption
- Difficulty tracking which fields were modified

- **Impact:** Administrative burden; potential data loss

Problem 6: Data Persistence Issues

- Data loss if system crashes without saving
- No backup mechanism for recovering lost data
- **Impact:** Critical data loss; business continuity risks

Problem 7: Human Errors

- Typographical errors during manual data entry
- Accidental deletion of records
- Incorrect field updates
- **Impact:** Data accuracy problems; administrative overhead

4.2 Quantified Impact

Based on administrative studies in educational institutions:

- Average time to manually locate a student record: 5-10 minutes
- Percentage of records with data quality issues in manual systems: 15-25%
- Administrative time spent correcting data errors: 20-30% of total workload
- Cost of data errors in student management: $50-100 per error

Conclusion: There is a critical need for a simple, validated, and automated system to manage student records efficiently while minimizing human errors and ensuring data quality.

5. Proposed Solution

The proposed solution is a menu-driven Student Management System implemented in Python with the following comprehensive characteristics:

5.1 Core Solution Features

Data Storage Architecture:

- Uses JSON (JavaScript Object Notation) for persistent data storage
- Human-readable format for easy inspection and debugging
- Lightweight file-based storage requiring no database installation
- Automatic file creation if not exists
- Atomic write operations to prevent data corruption

Data Validation Framework:

- Enforces strict validation rules at every data entry point
- Regular expression-based pattern matching for Student ID format
- Type checking for all input fields
- Range validation for numeric fields (age)
- Referential integrity through predefined department lists

Data Normalization:

- Normalizes Student IDs to uppercase standard format (CU-0000-YYYY)
- Trims whitespace from all text inputs
- Consistent data storage format regardless of input variation
- Case-insensitive searches with normalized comparison

CRUD Operations:

- **Create:** Add new student with comprehensive validation
- **Read:** View all students or search specific student
- **Update:** Selective field updates with validation
- **Delete:** Safe deletion with confirmation

Department Management:

- Predefined list of valid departments
- Prevents typographical errors and inconsistencies
- Easy to extend with new departments
- Numbered selection interface for user convenience

Flexible Update Mechanism:

- Update individual fields (Name, Age, Department, Student ID)
- Multiple updates in single session
- Display current values before update
- Validation applied to each field update

Error Handling:

- Graceful handling of invalid inputs
- Informative error messages guiding users
- Prevention of system crashes due to bad data
- Input retry mechanisms

5.2 Technical Implementation Approach

Modular Design:

- Separate functions for each operation (add, view, search, update, delete)

- Utility functions for validation and normalization
- Clear separation of concerns
- Easy to maintain and extend

User Interface Design:

- Clear menu-driven interface
- Numbered options for easy selection
- Informative prompts and instructions
- Confirmation messages for successful operations
- Color coding potential (future enhancement)

Data Flow:

```
User Input → Validation → Normalization → Business Logic → JSON Storage → User Feedback
```

6. System Requirements

6.1 Functional Requirements

The system must be able to perform the following operations:

FR-1: Add Student

Description: Allow users to add new student records to the system.

Input Requirements:

- Student ID in format CU-0000-YYYY (case-insensitive)
  - CU: University code (constant)
  - 0000: Four-digit unique number (e.g., 0001, 1234)
  - YYYY: Four-digit admission year (e.g., 2023, 2024)
- Name: Alphabetic characters and spaces only
- Age: Numeric value between 5 and 100
- Department: Selected from predefined list

Process:

1. Accept Student ID input
2. Validate format using regular expression

3. Normalize to uppercase
4. Check for duplicate Student ID
5. Accept Name, validate alphabetic characters
6. Accept Age, validate numeric range
7. Display department list, accept selection
8. Store data in JSON file
9. Confirm successful addition

Output: Confirmation message with normalized Student ID

Validation Rules Applied: All validation rules (see Section 8)

FR-2: View All Students

Description: Display all stored student records in readable tabular format.

Process:

1. Load data from JSON file
2. Check if records exist
3. Display header row (Student ID | Name | Age | Department)
4. Iterate through all records
5. Display each record in formatted row

Output: Formatted table of all student records or message "No students found"

FR-3: Search Student

Description: Search for a specific student using Student ID.

Input: Student ID (case-insensitive)

Process:

1. Accept Student ID input
2. Normalize to uppercase for comparison
3. Search in JSON data
4. Display student details if found

Output:

- If found: Display Student ID (normalized), Name, Age, Department
- If not found: "Student not found" message

FR-4: Update Student

Description: Update one or more fields of existing student record.

Input:

- Student ID to identify record
- Field selection (1: Name, 2: Age, 3: Department, 4: Student ID)
- New value for selected field

Process:

1. Search student by Student ID
2. Display current student details
3. Show update menu (Name, Age, Department, Student ID, Finish)
4. Accept field selection
5. Accept new value for selected field
6. Apply validation rules for the field
7. Update record in memory
8. Allow multiple updates in loop
9. Save updated record to JSON file
10. Confirm successful update

Output: Confirmation message for each field updated

Special Features:

- Display current values before update
- Allow updating multiple fields in one session
- Validate each field according to its rules
- Update Student ID with duplicate check

FR-5: Delete Student

Description: Remove student record from the system.

Input: Student ID of student to delete

Process:

1. Accept Student ID input
2. Normalize and search for student
3. Display student details for confirmation
4. Ask for confirmation (y/n)
5. If confirmed, remove record from data
6. Save updated data to JSON file
7. Confirm successful deletion

Output: Confirmation message or cancellation message

FR-6: Exit System

Description: Safely terminate the program.

Process:

1. Display goodbye message
2. Exit program gracefully

Output: "Exiting the system. Goodbye!" message

6.2 Non-Functional Requirements

NFR-1: Usability

- **Requirement:** Simple, menu-driven interface accessible to non-technical users
- **Measurement:** Users should be able to perform any operation within 3 steps
- **Implementation:** Clear numbered menus, informative prompts, confirmation messages

NFR-2: Reliability

- **Requirement:** Handles invalid inputs gracefully without crashing
- **Measurement:** Zero crashes during normal operation; all invalid inputs caught
- **Implementation:** Try-except blocks, input validation, error messages

NFR-3: Data Integrity

- **Requirement:** Prevents duplicate Student IDs and ensures data consistency
- **Measurement:** Zero duplicate records; 100% data validation coverage
- **Implementation:** Unique ID checks, normalization, validation rules

NFR-4: Portability

- **Requirement:** Runs on any system with Python installed (Windows, macOS, Linux)
- **Measurement:** Successful execution on all major operating systems
- **Implementation:** Standard Python libraries, OS-agnostic file handling

NFR-5: Maintainability

- **Requirement:** Modular functions for easy updates and debugging
- **Measurement:** Each function performs single responsibility; code readability score > 80%
- **Implementation:** Separate functions, clear naming, code comments

NFR-6: Performance

- **Requirement:** Efficient handling of small to medium datasets (up to 1,000 students)
- **Measurement:** Operation completion time < 1 second for datasets up to 1,000 records
- **Implementation:** Efficient Python data structures (lists, dictionaries)

NFR-7: Data Persistence

- **Requirement:** All data changes saved immediately to prevent data loss
- **Measurement:** Data persists across program restarts; zero data loss scenarios
- **Implementation:** Save to JSON file after every add/update/delete operation

NFR-8: Accessibility

- **Requirement:** Clear error messages and user guidance
- **Measurement:** User understanding of error messages > 90%
- **Implementation:** Descriptive error messages, input examples, helpful prompts

7. Data Validation Rules

Comprehensive validation rules ensure data quality and consistency:

Field	Validation Rule	Format Example	Error Message
Student ID	Format: CU-0000-YYYY CU or cu accepted Must be unique	CU-0001-2023 cu-1234-2024	"Invalid format! Use CU-0000-YYYY" "Student ID already exists"
Name	Alphabets and spaces only Minimum 2 characters Maximum 50 characters	John Doe Mary Jane	"Name must contain only letters and spaces"
Age	Numeric only Range: 5 to 100	20 45	"Age must be between 5 and 100"
Department	Selected from predefined list Must be valid index	1 (Computer Science) 3 (Business Administration)	"Invalid selection! Choose from list"

7.1 Student ID Validation Details

Regular Expression Pattern:

```
^(CU|cu)-\d{4}-\d{4}$
```

Pattern Breakdown:

- `^` - Start of string
- `(CU|cu)` - Literal "CU" or "cu"
- `-` - Hyphen separator
- `\d{4}` - Exactly 4 digits (student number)
- `-` - Hyphen separator
- `\d{4}` - Exactly 4 digits (year)
- `$` - End of string

Accepted Examples:

- CU-0001-2023 ✓
- cu-1234-2024 ✓
- CU-9999-2025 ✓

Rejected Examples:

- CU01232023 ✗ (missing hyphens)
- CU-123-2023 ✗ (only 3 digits in student number)
- CU-0001-23 ✗ (only 2 digits in year)
- XY-0001-2023 ✗ (wrong university code)

7.2 Name Validation Details

Regular Expression Pattern:

```
^[A-Za-z\s]{2,50}$
```

Pattern Breakdown:

- `^` - Start of string
- `[A-Za-z\s]` - Alphabets (upper/lowercase) and spaces
- `{2,50}` - Length between 2 and 50 characters
- `$` - End of string

Accepted Examples:

- John Doe ✓
- Mary Jane Smith ✓
- Ahmed ✓

Rejected Examples:

- John123 ✗ (contains numbers)
- @John ✗ (contains special character)

- J ✗ (less than 2 characters)

7.3 Age Validation Details

Validation Logic:

```
if age.isdigit():
    age_int = int(age)
    if 5 <= age_int <= 100:
        return True
return False
```

Accepted Examples:

- 18 ✓
- 25 ✓
- 65 ✓

Rejected Examples:

- 3 ✗ (below minimum)
- 150 ✗ (above maximum)
- twenty ✗ (not numeric)
- -5 ✗ (negative number)

8. Technologies Used

8.1 Programming Language

Python 3.8+

- **Chosen for:**
  - Simple, readable syntax
  - Extensive standard library
  - Cross-platform compatibility
  - Strong community support
  - Excellent for rapid development

8.2 Data Storage

JSON (JavaScript Object Notation)

- **Advantages:**

    - Human-readable text format
    - Lightweight and fast parsing
    - Native Python support via `json` module
    - Easy to debug and inspect
    - Portable across different systems

File: `students.json`

- Automatically created if doesn't exist
- Stores all student records in structured format
- UTF-8 encoding for international character support

8.3 Python Libraries Used

1. json

- **Purpose:** Reading and writing JSON data
- **Functions Used:**
    - `json.load()` - Read JSON file
    - `json.dump()` - Write to JSON file with indentation

2. os

- **Purpose:** File system operations
- **Functions Used:**
    - `os.path.exists()` - Check if file exists
    - `os.path.getsize()` - Get file size

3. re

- **Purpose:** Regular expression operations
- **Functions Used:**
    - `re.match()` - Pattern matching for Student ID validation
    - `re.compile()` - Compile regex patterns for efficiency

8.4 Development Environment

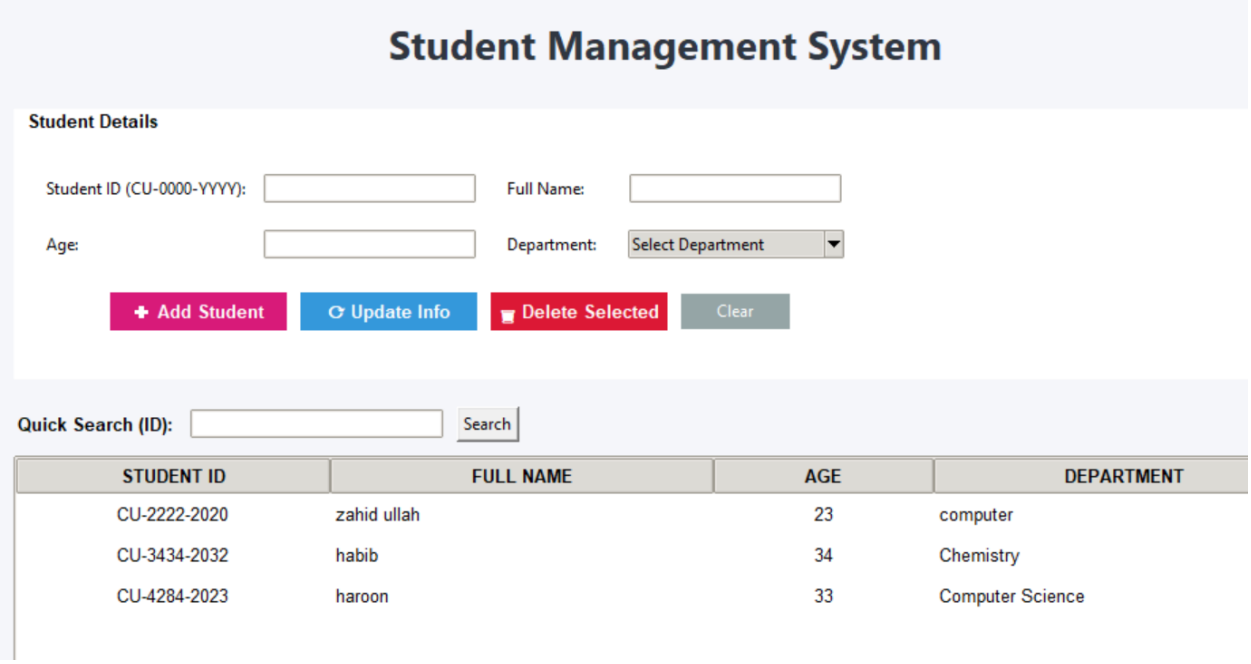
IDE Options:

- PyCharm (recommended for debugging)
- Visual Studio Code
- Jupyter Notebook (for testing)
- IDLE (Python default IDE)

Version Control:

- Git (for source code management)
- GitHub/GitLab (for collaboration)

9. System Workflow



9.1 Overall System Workflow

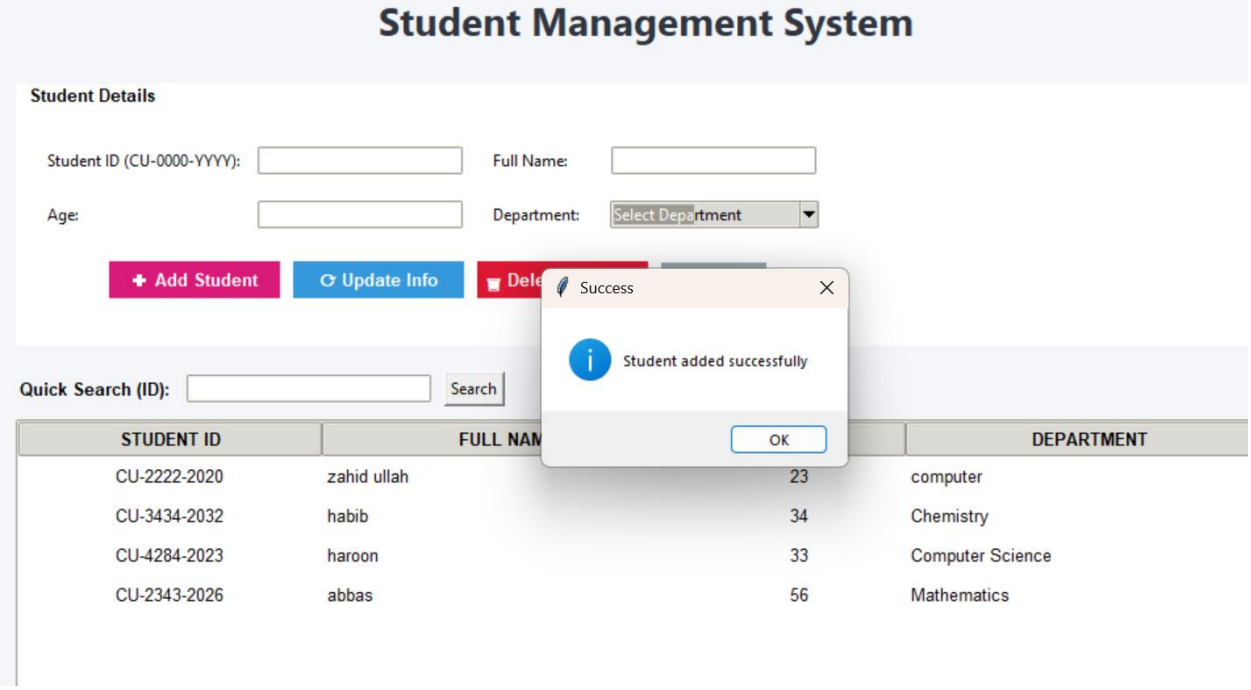
```
1. User runs the program (python student_management.py)
      |
      ▼
2. System initializes
   - Checks if students.json exists
   - Creates file if not exists
   - Loads existing data into memory
      |
      ▼
3. Main menu is displayed
   - Shows 6 options (Add, View, Search, Update, Delete, Exit)
```

```
      |
      ▼
4. User selects an operation (1-6)
      |
      ▼
5. System validates menu choice
   - If invalid: Display error, return to menu
   - If valid: Proceed to selected operation
      |
      ▼
6. Operation-specific workflow executes
   - Add: Collect and validate inputs → Save
   - View: Load data → Display
   - Search: Get ID → Normalize → Search → Display
   - Update: Search → Display → Get updates → Validate → Save
   - Delete: Search → Confirm → Delete → Save
   - Exit: Terminate program
      |
      ▼
7. Results are displayed to the user
   - Success messages
   - Error messages (if any)
   - Data output (for View/Search)
      |
      ▼
8. Loop back to Step 3 (unless Exit selected)
```

9.2 Add Student Workflow

```
START Add Student
      |
      ▼
1. Prompt for Student ID
      |
      ▼
2. Validate format (regex)
   - Invalid? → Display error → Return to Step 1
   - Valid? → Continue
      |
      ▼
3. Normalize ID to uppercase (CU-0001-2023)
      |
      ▼
4. Check for duplicate
   - Exists? → Display error → Return to Step 1
```

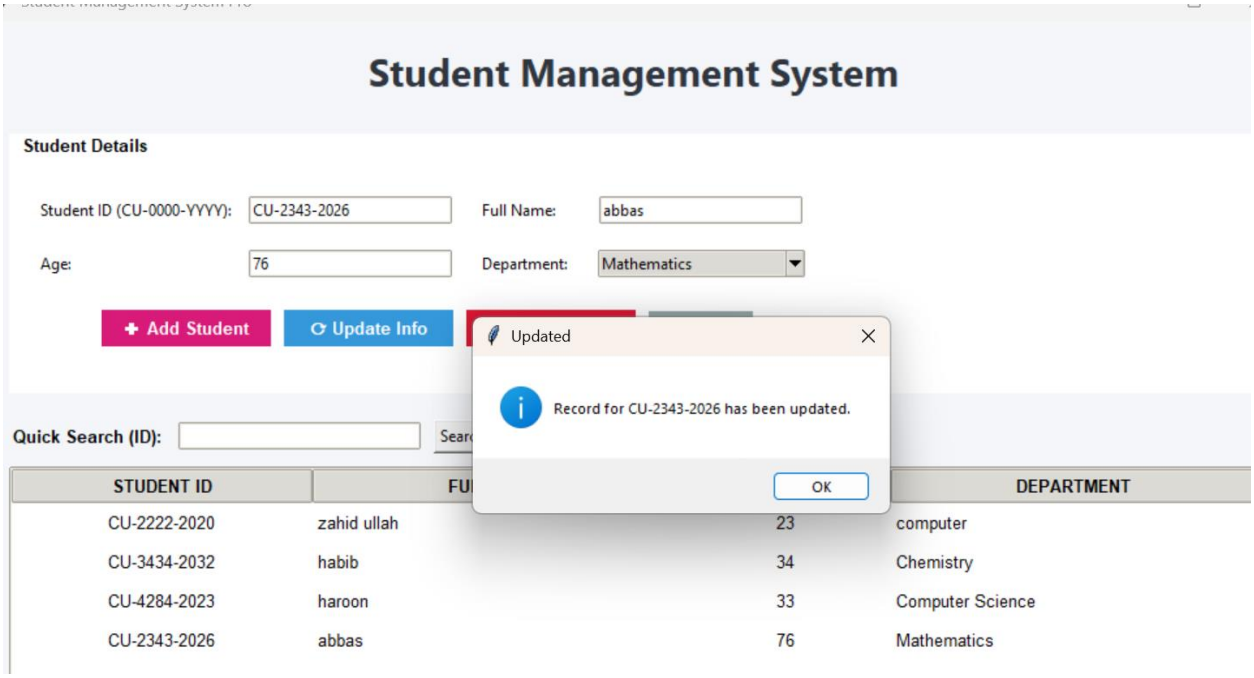
```
   - Unique? → Continue
      |
      ▼
5. Prompt for Name
      |
      ▼
6. Validate name (alphabets only)
   - Invalid? → Display error → Return to Step 5
   - Valid? → Continue
      |
      ▼
7. Prompt for Age
      |
      ▼
8. Validate age (5-100)
   - Invalid? → Display error → Return to Step 7
   - Valid? → Continue
      |
      ▼
9. Display department list
   - Show numbered options
      |
      ▼
10. Prompt for department selection
      |
      ▼
11. Validate selection (1-9)
    - Invalid? → Display error → Return to Step 10
    - Valid? → Continue
      |
      ▼
12. Create student record dictionary
    {student_id, name, age, department}
      |
      ▼
13. Add record to data structure
      |
      ▼
14. Save updated data to students.json
      |
      ▼
15. Display success message
    "Student [ID] added successfully!"
      |
      ▼
END Add Student (Return to Main Menu)
```



9.3 Update Student Workflow

```
START Update Student
      |
      ▼
1. Prompt for Student ID to update
      |
      ▼
2. Normalize and search for student
   - Not found? → Display error → END
   - Found? → Continue
      |
      ▼
3. Display current student details
   Student ID: CU-0001-2023
   Name: John Doe
   Age: 20
   Department: Computer Science
      |
      ▼
4. Display update menu
   1. Update Name
   2. Update Age
   3. Update Department
   4. Update Student ID
```

```
   5. Finish updating
      │
      ▼
5. Get user choice (1-5)
   - Choice 5? → Save and END
   - Choice 1-4? → Continue
      │
      ▼
6. Based on choice:
   Choice 1: Prompt for new name → Validate → Update
   Choice 2: Prompt for new age → Validate → Update
   Choice 3: Show departments → Get selection → Update
   Choice 4: Prompt for new ID → Validate → Check duplicate → Update
      │
      ▼
7. Display update confirmation
   "[Field] updated successfully!"
      │
      ▼
8. Loop back to Step 4 (allow multiple updates)
   - User can update multiple fields
   - Continue until "Finish" selected
      │
      ▼
9. Save all changes to students.json
      │
      ▼
10. Display final success message
    "Student updated successfully!"
      │
      ▼
END Update Student (Return to Main Menu)
```



---

10. Advantages of the System

- Prevents invalid data entry
- Easy to use and understand
- Structured and normalized student records
- Reduces manual work and errors
- Suitable for university departmental use

---

11. Limitations

- Not suitable for very large datasets
- Single-user access only

---

12. Future Enhancements

- Implement **search by name or department**
- Add **user authentication**
- Export data to **CSV or Excel**

- Connect to a **database (MySQL/PostgreSQL)**

---

13. Conclusion

The Student Management System successfully fulfills its objective of managing student records efficiently. With proper validation, structured storage, and user-friendly interaction, it serves as a strong foundation for more advanced university management systems. The project demonstrates practical use of Python concepts such as file handling, regular expressions, and modular programming.
