



# Student Management

26.04.25

— *Jazlin Mazumder* 240005849

## Overview

The Student Management System (SMS) is a Python-based application developed to manage student records using Object-Oriented Programming (OOP) principles. It supports CRUD operations—Create, Read, Update, and Delete—for both regular and undergraduate students. Data is persistently stored in JSON format, allowing the program to retain information across sessions. The system emphasises modular design, data validation, exception handling, and user-friendly interaction via a command-line menu interface.

## Goals

1. Apply key OOP concepts such as encapsulation and inheritance to real-world scenarios.
2. Design reusable, modular, and maintainable code through functions and class-based design.
3. Persistently store student data in JSON formats.
4. Validate user inputs for data integrity, including student ID uniqueness and valid course/year values.
5. Implement robust error handling using custom exceptions.
6. Provide an interactive, guided user interface for effective use of the system.

## Specifications

This system was structured with modular components and a clear file hierarchy to separate concerns and simplify maintenance. Key features and technical specifications include:

### Core Features

- **CRUD Operations:** Users can add, update, read, and delete student records through a structured menu system.
- **Data Storage:** Student records are stored in JSON files using the `storage.py` module to support data persistence.
- **Validation:** User input is validated using the `validation.py` module, ensuring correct ID format, course names, and year values. Duplicate student IDs are prevented.
- **Custom Exceptions:** Robust exception handling is implemented using custom-defined exceptions such as `InvalidIDException`, `DuplicateStudentIDException`.
- **OOP Design:**
  - The `Student` class encapsulates sensitive data like the student ID.
  - The `Undergraduate` subclass inherits from `Student` and adds attributes like `minor` specialisation.
- **User Interaction:** The `main.py` script provides a menu-driven interface allowing users to perform operations interactively.

### File Structure

- `main.py`: Entry point for launching the interactive system.

- `student_operations.py`: Contains core CRUD logic and data manipulation functions.
- `database.py`: In-memory data storage for active student records.
- `validation.py`: Functions to validate student attributes (ID, year, course, etc.).
- `storage.py`: Handles reading from and writing to JSON files.
- `exceptions.py`: Custom exception classes for input and logic validation.
- `models/student.py`: Base `Student` class implementing encapsulation.
- `models/undergraduate.py`: `Undergraduate` class extending `Student`.

## Implementation Details

- **Encapsulation:** The `Student` class secures the `__student_id` attribute using getter and setter methods.
- **Inheritance:** `Undergraduate` extends `Student` with a `minor` field and overrides the `get_details()` method using `super()`.
- **Menu System:** The application runs in an interactive loop, allowing users to select options like add, delete, update, and list students. Each operation includes data validation and error handling.
- **Persistence:** Data is stored in `students.json`, allowing students to be reloaded automatically when the program starts.

## Usage

To run the application:

1. Ensure Python 3.8+ is installed (tested on Python 3.12).
2. Extract the project folder and navigate to it in your terminal.

3. Once running Main.py, the interactive menu allows users to:
  1. Add a student
  2. Delete a student
  3. Update a student
  4. List all students
  5. Exit the programAll operations are guided, and no additional setup is required.

## Error Handling

The system uses custom exceptions to manage various edge cases and input errors, ensuring the system does not crash and provides meaningful feedback:

- **InvalidIDException**: Triggered when a student ID format is incorrect.
- **DuplicateStudentIDException**: Prevents adding students with duplicate ID's.

## Milestones

### Initial Development

- Created **Student** and **Undergraduate** classes with encapsulated fields and inheritance.
- Built out file structure for modular code organisation.

### Feature Implementation

- Implemented all CRUD functions and interactive menu in **main.py**.

- Built `validation.py` for data integrity checks.
- Developed custom exceptions in `exceptions.py` to handle all expected input errors.

## Data Handling and Testing

- Enabled persistent storage using JSON format via `storage.py`.
- Fully tested menu-driven system with realistic inputs and simulated edge cases.

## Challenges Faced and Solutions Applied:

### Validating edge cases (e.g., invalid ID's or years)

Developed a dedicated `validation.py` module and used custom exceptions such as `InvalidIDException`

### Handling duplicate student entries:


Integrated validation to check for existing ID's and raised `DuplicateStudentIDException` when duplicates were found.

### Complexity in subclassing and data compatibility:

Used inheritance (`Undergraduate` subclassing `Student`) and ensured `storage.py` serialised both object types correctly.

## Recommendations for Future Improvements

- **GUI Implementation:** Add a graphical user interface (GUI) using `Tkinter` or `PyQt` for a more user-friendly experience.
- **Database Integration:** Replace or supplement file storage with a lightweight database like SQLite for better scalability.

- 
- **Search and Filter Functionality:** Implement a filtering mechanism that allows users to search for students by year, course, or type.
  - **Role-Based Access Control:** Add admin vs. user functionality to restrict access to certain features.
  - **Data Encryption:** Encrypt sensitive data, such as student ID's, before storage for enhanced security.

# Data Cleansing

26.04.25

## Overview

This project focuses on processing, analysing, and visualising real-world, messy data using Python and data science libraries such as **Pandas**, **NumPy**, **Matplotlib**, **Seaborn**, and **Scikit-learn**. It simulates challenges common in real-life datasets, including missing values, inconsistent formatting, duplicates, and outliers.

The objective was to clean a sales dataset containing over 5,500 rows, extract meaningful insights through exploratory data analysis (EDA), generate visual representations of trends, and optionally implement advanced techniques such as clustering, regression, and interactive dashboards for real-time filtering.

## Goals

1. Clean and preprocess messy real-world data using Pandas.
2. Transform and wrangle data for analysis using aggregation, filtering, and transformation techniques.
3. Perform exploratory data analysis (EDA) to extract statistical insights and discover correlations.
4. Create visualisations using Matplotlib and Seaborn to enhance data comprehension.
5. Implement interactive dashboards that allow user-driven exploration of data.
6. Apply advanced data analysis techniques like clustering and regression for deeper insight.

## Specifications

This project is organised into multiple Python modules, each focusing on a specific stage of the data analysis pipeline, from cleaning to advanced analysis. The code follows modular principles and is well-documented for maintainability.

### Core Features

- **Data Cleaning and Preprocessing**
  - Missing values are handled using median, mean, or mode strategies.
  - Duplicate records identified and removed.
  - Data type inconsistencies resolved (e.g., date strings to `datetime`).
  - Outliers were detected and managed using statistical methods (IQR).
- **Data Wrangling and Transformation**
  - Filtering and aggregation are performed using `sum()`, and other Pandas functions.
  - Lambda functions are used for column-level transformations.
  - A new `Profit Margin` column is computed using `Profit / Sales`.
- **Exploratory Data Analysis (EDA)**
  - Summary statistics generated via `.describe()`.
  - Correlations between numerical variables are explored using correlation matrices.



- Pivot tables are used to analyse trends across categories and discount levels.
- **Data Visualisation**
  - Various plot types created: bar charts, histograms, scatter plots, and box plots.
  - Visualisations customised with titles, labels, legends, and colour palettes.
  - Seaborn is used for enhanced styling and aesthetic presentation.
- **Interactive Dashboard (Optional)**
  - Allows users to filter data by category and dynamically generate visualisations based on user input.
  - Users can select chart types (e.g., barplot) for interactive data exploration.
- **Advanced Data Analysis (Optional)**
  - KMeans clustering is implemented to group customers based on numerical features.
  - Missing sales values are predicted using a linear regression model.
  - Heatmaps were generated to visualise correlations among features.

## File Structure

- **dataCleaningAndPreprocessing.py**: Handles missing values, duplicates, data types, and outliers.
- **dataWranglingAndTransformation.py**: Performs grouping, filtering, transformations, and adds new calculated columns.
- **exploratoryDataAnalysis.py**: Generates statistical summaries, correlations, and pivot tables.

- `dataVisualisation.py`: Creates and customises plots using Seaborn and Matplotlib.
- `interactiveDashboardForDataFiltering.py`: Implements real-time user-driven filtering and visualisation.
- `advancedDataAnalysis.py`: Conducts clustering, regression, and correlation heatmaps.
- `messy_dataset.csv`: The original, unprocessed dataset.
- `data-dropna.csv`: Cleaned and structured dataset ready for analysis.

## Implementation Details

- **Data Cleaning:**
  - Used `pd.read_csv()` to import the dataset.
  - Missing numeric values filled using `df.fillna(df.median())`.
  - Duplicates dropped with `df.drop_duplicates()`.
  - Dates parsed with `pd.to_datetime()`.
  - Outliers removed using the IQR method.
- **Transformation:**
  - Grouped sales data by category to produce aggregated summaries.
  - Created a `Profit Margin` field using a lambda function:
- **EDA Techniques:**
  - `df.describe()` to generate overall statistics.
  - `df.corr()` to compute the correlation matrix.

- **Interactivity:**

- Took user input via `input()` to filter by category.
- Dynamically generated barplots based on the filtered dataset.

- **Advanced Analysis:**

- Used `sklearn.cluster.KMeans` for clustering customer segments.
- Built a linear regression model using `sklearn.linear_model.LinearRegression` to predict missing sales values.

## Usage

1. **Install Python 3.8+** (Tested on Python 3.12).
2. **Install Imports:**
  1. `pip install pandas`
  2. `pip install matplotlib`
  3. `pip install seaborn`
  4. `pip install scikit-learn`
3. **Run the Scripts Individually:** Each `.py` file is modular and can be run independently, depending on which stage of analysis is needed.
4. **Interactive Execution:** Use `interactiveDashboardForDataFiltering.py` to explore and visualise data dynamically based on your input filters.

## Error Handling and Data Quality

- Missing values are carefully imputed using statistical methods.
- All operations check for data integrity (e.g., ensuring no divide-by-zero during calculations).
- Outliers are removed to avoid skewing analysis results.

- Conversions and imputations are wrapped with exception-handling mechanisms where applicable.

## Milestones

### Initial Setup

- Loaded messy dataset and diagnosed common issues like nulls, inconsistent types, and duplicates.

### Core Development

- Completed cleaning and transformation modules.
- Integrated statistical functions and transformations for new insights.

### EDA and Visualisation


- Generated meaningful summaries, pivot tables, and correlation heatmaps.
- Produced clear, visually appealing graphs to identify key insights.

### Bonus Implementation

- Created interactive filtering with user-driven chart generation.
- Conducted clustering and regression analysis using scikit-learn.

## Challenges Faced and Solutions Applied:

**Outliers skewing analysis:**



Detected outliers using IQR method and removed them to ensure meaningful statistical representation.

### **Performance bottlenecks due to large dataset size:**

Processed the data in batches during initial EDA and used `.copy()` to avoid chained assignments and memory leaks.

### **User errors in interactive inputs (e.g., misspelling category names)**

Implemented checks to validate user input and fallback messages for unsupported entries

## **Recommendations for Future Improvements**

- **Dashboard Upgrade:** Use **Streamlit** or **Dash** instead of `input()` for a richer, GUI-based dashboard experience.
- **Unit Testing:** Add automated unit tests for each transformation and cleaning function to ensure robustness and reproducibility.
- **Improved Error Logging:** Include a logging module to track failed operations, especially in the preprocessing phase.
- **Documentation and Notebook Version:** Provide an interactive **Jupyter Notebook** version for teaching, demonstration, or presentation purposes.
- **Database Integration:** Store and query cleaned data using a lightweight database such as SQLite or PostgreSQL for scalability.

