**Introduction:**

* The Match Result Manager class is a Python implementation designed to manage the recording, retrieval, and analysis of match results in competitive sports or gaming contexts. This class provides an efficient way to track the outcomes of various matches, allowing users to manage game results and analyse team performance over time.

MATCH RESULT MANAGER

**Existing System:**

* **Manual Record-Keeping**: Many leagues or tournaments still rely on spreadsheets or paper-based systems to track match results, leading to potential errors and inconsistencies.
* **Limited Accessibility**: Traditional systems may not provide easy access to match data, requiring users to sift through numerous files or documents to find information.
* **Lack of Analysis Tools**: Many existing systems do not offer robust analytical capabilities, making it difficult for teams and coaches to evaluate performance metrics effectively.

**Proposed System:**

* **User-Friendly Interface**:

Can be integrated into a simple user interface that allows easy input of match data.

* **Quick Access**:

Users can retrieve match results using a unique match\_id, streamlining the process of accessing historical historical data.

* **Dynamic Data Management**:

Users can easily update or delete match results without extensive manual intervention, ensuring the data remains accurate and current.

VISION:

To provide an intuitive and efficient tool for managing and analyzing sports match results, empowering teams, coaches, and fans with actionable insights to enhance performance and engagement.

MISSION:

Our mission is to develop a reliable and user-friendly match management system that:

* Facilitates seamless recording and retrieval of game outcomes.
* Promotes fair play and transparency by allowing users to track match history and statistics.

**Requirements Specification:**

**Functional Requirements**

1. **Record Game Outcomes**:

The system must allow users to record the outcomes of matches by specifying a unique match ID, team names, and scores.

1. **Update Existing Matches**:

The system must enable users to update the outcomes of an existing match if the match ID is already recorded.

1. **Delete Game Results**:

The system must provide a mechanism to delete the results of a specific match using its match ID.

**Non-Functional Requirements**

1. **Usability**:

The system should be easy to use, with a clear interface for recording, retrieving, and analyzing match data.

1. **Performance**:

The system should perform operations (record, update, delete, retrieve, analyze) efficiently, even with a large number of matches.

1. **Security**:

While not directly applicable in this context, the system should ensure that sensitive data (if any) is handled securely, and operations are restricted to authorized users if implemented in a multi-user environment.

Requirements Specification:

**System Requirements**

* + - **Software Requirements**

- Python 3.2 Functional

- Vs Code

- SQLite

* + - **Hardware Requirements**

- Processor : Intel i3

- Hard disk : 500GB

- RAM : 8GB

**Module and Classes**:

**Match Result Manager**:

This class manages multiple Match instances. It provides methods to record game outcomes, delete results, retrieve results, analyze team performance, and display match results

**Source Code :**

class Match:

    def \_\_init\_\_(self, match\_id, team1, team2, score1, score2):  # Corrected the constructor

        self.match\_id = match\_id

        self.team1 = team1

        self.team2 = team2

        self.score1 = score1

        self.score2 = score2

class MatchResultManager:

    def \_\_init\_\_(self):  # Corrected the constructor

        self.matches = {}

    def record\_game\_outcomes(self, match\_id, team1, team2, score1, score2):

        if score1 < 0 or score2 < 0:

            raise ValueError("Scores must be non-negative.")

        if not team1 or not team2:

            raise ValueError("Team names must not be empty.")

        self.matches[match\_id] = Match(match\_id, team1, team2, score1, score2)

    def delete\_game\_result(self, match\_id):

        if match\_id in self.matches:

            del self.matches[match\_id]

        else:

            raise KeyError(f"No match found with ID: {match\_id}")

    def retrieve\_game\_result(self, match\_id):

        match = self.matches.get(match\_id)

        if match is None:

            raise KeyError(f"No match found with ID: {match\_id}")

        return vars(match)

    def analyze\_team\_performance(self, team\_id):

        performance = {'games\_played': 0, 'wins': 0, 'draws': 0, 'losses': 0}

        for match in self.matches.values():

            if team\_id in (match.team1, match.team2):

                performance['games\_played'] += 1

                if (team\_id == match.team1 and match.score1 > match.score2) or \

                   (team\_id == match.team2 and match.score2 > match.score1):

                    performance['wins'] += 1

                elif match.score1 == match.score2:

                    performance['draws'] += 1

                else:

                    performance['losses'] += 1

        return performance

    def display\_match\_results(self):

        for match in self.matches.values():

            print(f"Match ID: {match.match\_id} | {match.team1} {match.score1} - {match.score2} {match.team2}")

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

    manager = MatchResultManager()

    manager.record\_game\_outcomes("001", "Team A", "Team B", 3, 1)

    manager.record\_game\_outcomes("002", "Team A", "Team B", 0, 0)

    manager.display\_match\_results()

    print(manager.analyze\_team\_performance("Team B"))

**Benefits:**

Persistence of Data

* Serialization: The ability to save and load match data from a file ensures that data is not lost between sessions. This is particularly useful for applications that need to retain data over time.

User-Friendly Interface

* Interactive User Interface: A simple command-line interface allows users to interact with the program easily, record matches, and analyze results without needing to modify the code directly.

Modular Design

* Separation of Concerns: The code is organized into classes and methods with clear responsibilities. This modularity makes the code easier to understand, test, and maintain

**Conclusion :**

Overall, these enhancements result in a more robust, user-friendly, and maintainable system that can efficiently manage match results and provide valuable insights into team performance. This makes the code suitable for a wide range of applications, from personal projects to more formal sports management tools

**Future Enhancements :**

**Serialization Support:**

Implement methods to save and load match data to/from a file (e.g., JSON format) for persistence.

**For Example :**

Code:

import json

def save\_to\_file(self, filename):

with open(filename, 'w') as file:

json.dump([vars(match) for match in self.matches.values()], file)

def load\_from\_file(self, filename):

with open(filename, 'r') as file:

matches = json.load(file)

for match in matches:

self.matches[match['match\_id']] = Match(\*\*match)

**For Reference :**

1. Fluent Python: Clear, Concise, and Effective Programming

**By Luciano Ramalho**

1. The Python Book

**By Rob Mastrodomenico**