LAB TEST-02

Name : P.Reshmitha Reddy

Rollno : 2403A510A7

Batch : 05

Course : AI Assisted Coding

Subgroup A
A.1 — [S09A1] Compute per-player average from logs (AI completion)
Scenario (sports analytics):
Context:
You are integrating a sports analytics telemetry service where each player emits periodic
measures as CSV lines: `id,timestamp,velocity`. Due to flaky connectivity, some lines may be
truncated or contain non-numeric values. Ops needs a quick aggregation for dashboards and
alert thresholds.
Your Task:

Write a Python function to parse the raw text (multiple lines) and compute per-player averages

of `velocity` . Return a dict {id: avg} and separately compute an overall average.
Data & Edge Cases:
Input contains newlines, optional leading/trailing spaces, and may include malformed rows.
Timestamps are ISO-8601 but not needed for math.
AI Assistance Expectation:
Use AI code completion to scaffold the loop, dictionary accumulation (sum and count), and
exception handling for malformed rows.
Constraints & Notes:
Prefer O(n) pass; ignore lines that cannot be split into three fields or have non-numeric metric;
round averages to 2 decimals.
Sample Input
pl91,2025-01-01T08:00,28.7
pl92,2025-01-02T09:00,30.2
pl93,2025-01-03T010:00,31.7
Sample Output
{'pl91': 28.7, 'pl92': 30.2, 'pl93': 31.7} and overall_avg=30.2
Acceptance Criteria: Correct averages per ID; overall average reported; malformed lines skipped

Prompt :# Write a Python function that parses raw CSV lines of id,timestamp,velocity, skips malformed or non-numeric rows, and returns per-player and overall average velocities rounded to 2 decimals.

Code :

```python
def compute_player_averages(raw_text):
    """
    Parses raw CSV telemetry logs and computes per-player and overall velocity averages.
    Skips malformed lines and non-numeric velocities.
    Returns: (dict of {id: avg_velocity}, overall_avg)
    """
    from collections import defaultdict

    sums = defaultdict(float)
    counts = defaultdict(int)
    total_sum = 0.0
    total_count = 0

    for line in raw_text.strip().splitlines():
        line = line.strip()
        if not line:
            continue
        parts = line.split(',')
        if len(parts) != 3:
            continue
        player_id, _, velocity_str = parts
        try:
            velocity = float(velocity_str)
```

```
            except ValueError:
                continue
            sums[player_id] += velocity
            counts[player_id] += 1
            total_sum += velocity
            total_count += 1

    averages = {pid: round(sums[pid] / counts[pid], 2) for pid in sums}
    overall_avg = round(total_sum / total_count, 2) if total_count else 0.0
    return averages, overall_avg

# Sample usage
sample_input = """
pl91,2025-01-01T08:00,28.7
pl92,2025-01-02T09:00,30.2
pl93,2025-01-03T010:00,31.7
"""
result, overall = compute_player_averages(sample_input)
print(result)       # {'pl91': 28.7, 'pl92': 30.2, 'pl93': 31.7}
print("overall_avg=", overall)  # overall_avg= 30.2
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
{'pl91': 28.7, 'pl92': 30.2, 'pl93': 31.7}
overall_avg= 30.2
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The function efficiently parses raw CSV telemetry logs, handling multiple lines and skipping malformed or non-numeric entries.It uses a single pass (O(n)) to accumulate the sum and count of velocities per player, as well as overall totals.The use of defaultdict simplifies accumulation for each player ID.Averages are rounded to two decimal places for both per-player and overall results.The function is robust against empty lines, missing fields, and invalid data, making it suitable for real-world, noisy telemetry input.The output is

a dictionary mapping player IDs to their average velocities, along with the overall average, matching the requirements.

A.2 — [S09A2] Implement MatchStats with add/remove/summary (AI completion)
Scenario (sports analytics):
Context:
A microservice in the sports analytics platform maintains a small in-memory structure to track
values keyed by identifier (e.g., order IDs, sensor IDs). Engineers want a minimal class to add,
remove, and summarize current values for quick health checks.

Your Task:
Implement a `MatchStats` class with methods `add(id: str, value: float)`, `remove(id: str)`, and
`summary() -> tuple[int, float|None]` returning (count, average).
Data & Edge Cases:
IDs are unique keys. Re-adding the same ID overwrites its value. Removing a missing ID should
be safe (no exception). For an empty store, average is None.
AI Assistance Expectation:
Ask AI to generate the class skeleton with docstrings and

type hints, then refine method
behavior and add a quick usage example.
Constraints & Notes:
Keep state in a dict; O(1) per operation; return rounded average to 2 decimals (when non-
empty).
Sample Input
[{'op': 'add', 'id': 'a1', 'value': 18}, {'op': 'add', 'id': 'b2', 'value': 17}, {'op': 'remove', 'id': 'a1'}, {'op': 'add', 'id':
'c3', 'value': 15}]
Sample Output
count=2, avg=16.0
Acceptance Criteria: Handles add/remove; correct count and average; safe on missing IDs

Prompt :# Implement a MatchStats class with add(id, value), remove(id), and summary() methods to track unique IDs and compute the count and average (rounded to 2 decimals), handling missing IDs safely.

Code :

```python
class MatchStats:
    """
    Tracks values by unique ID with add, remove, and summary operations.
    """

    def __init__(self):
        """Initialize the internal store."""
        self._data: dict[str, float] = {}

    def add(self, id: str, value: float) -> None:
        """
        Add or update the value for a given ID.
        """
        self._data[id] = value

    def remove(self, id: str) -> None:
        """
        Remove the value for a given ID. Does nothing if ID is missing.
        """
        self._data.pop(id, None)

    def summary(self) -> tuple[int, float | None]:
        """
        Return a tuple of (count, average value rounded to 2 decimals).
        If no values, average is None.
        """
        count = len(self._data)
        if count == 0:
            return (0, None)
        avg = round(sum(self._data.values()) / count, 2)
        return (count, avg)

# Usage example
stats = MatchStats()
stats.add('a1', 18)
stats.add('b2', 17)
stats.remove('a1')
stats.add('c3', 15)
count, avg = stats.summary()
print(f"count={count}, avg={avg}")
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py

count=2, avg=16.0
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The MatchStats class uses a dictionary to store values keyed by unique IDs, ensuring O(1) time complexity for add, remove, and summary operations.The add method inserts or updates the value for a given ID.The remove method safely deletes an ID if present; if the ID is missing, it does nothing (no exception is raised).The summary method returns a tuple with the count of current entries and the average of their values (rounded to 2 decimals). If the store is empty, the average is None.The class is minimal, robust, and suitable for quick health checks or aggregations in a microservice context.The usage example demonstrates correct handling of add, remove, and summary operations, including safe removal of missing IDs.