AI ASSISSTED LAB EXAM-2

NAME:P.AJAY
ROLL.NO:2403A510B4
BATCH:05

A.1 — [S09A1] Compute per-player average from logs

Prompt

Write a Python function to parse raw text logs (id, timestamp, velocity) and compute per-player averages.
Handle malformed rows and non-numeric values by skipping them.
Return a dictionary of per-player averages and an overall average.

Code

```
from typing import Dict, Tuple

def compute_player_averages(log_text: str) -> Tuple[Dict[str, float], float]:
    """
    Parse raw CSV lines of format id,timestamp,velocity and compute per-player averages.
    """
    player_stats: Dict[str, Tuple[float, int]] = {}  # {id: (sum, count)}
    total_sum, total_count = 0.0, 0

    for line in log_text.strip().splitlines():
        line = line.strip()
        if not line:
            continue

        parts = line.split(",")
        if len(parts) != 3:
            continue  # malformed row, skip

        pid, _, vel = parts
        try:
            v = float(vel)
        except ValueError:
            continue  # skip non-numeric velocity

        if pid not in player_stats:
            player_stats[pid] = [0.0, 0]

        player_stats[pid][0] += v
        player_stats[pid][1] += 1

        total_sum += v
        total_count += 1

    # compute averages
    avg_dict = {pid: round(s / c, 2) for pid, (s, c) in player_stats.items() if c > 0}
    overall_avg = round(total_sum / total_count, 2) if total_count > 0 else 0.0

    return avg_dict, overall_avg


# ---- Sample Test ----
sample = """
p191,2025-01-01T08:00,28.7
p192,2025-01-02T09:00,30.2
p193,2025-01-03T010:00,31.7
```

Output

```
({'p191': 29.0, 'p192': 30.2, 'p193': 31.7}, 29.97)
PS C:\Users\AJAY\ai coding>
```

Observation

- Each valid line was parsed and accumulated.
- Malformed (`bad,line`) was skipped safely.
- Per-player averages were computed correctly and rounded.
- The overall average matches expected values.

A.2 — [S09A2] Implement MatchStats

Prompt

Implement a MatchStats class that supports:
- add(id: str, value: float) → insert or overwrite a value.
- remove(id: str) → safely delete an identifier.
- summary() → return (count, average); if empty, average = None.

Maintain O(1) performance using a dictionary.

Code

```
from typing import Dict, Tuple, Optional

class MatchStats:
    """
    Minimal in-memory store for tracking values keyed by identifier.
    Supports add, remove, and summary operations.
    """

    def __init__(self):
        self.data: Dict[str, float] = {}

    def add(self, id: str, value: float) -> None:
        """Add or update a value by key"""
        self.data[id] = value

    def remove(self, id: str) -> None:
        """Remove a key safely (ignore if missing)"""
        self.data.pop(id, None)

    def summary(self) -> Tuple[int, Optional[float]]:
        """Return (count, average) of current values; average=None if empty"""
        if not self.data:
            return (0, None)

        values = list(self.data.values())
        avg = round(sum(values) / len(values), 2)
        return (len(values), avg)


# ---- Sample Test ----
ops = [
    {'op': 'add', 'id': 'a1', 'value': 18},
    {'op': 'add', 'id': 'b2', 'value': 17},
    {'op': 'remove', 'id': 'a1'},
    {'op': 'add', 'id': 'c3', 'value': 15}
]

ms = MatchStats()
for op in ops:
    if op['op'] == 'add':
        ms.add(op['id'], op['value'])
    elif op['op'] == 'remove':
        ms.remove(op['id'])

print(ms.summary())
```

Output

Observation

- Values are stored in a dictionary for O(1) lookup/update/remove.
- Removing a missing key does not raise errors.
- Re-adding overwrites correctly.
- Summary shows correct count and average (rounded).
- For empty store, (0, None) is returned safely.