AI ASSISTED CODING

LAB EXAM – 2

NAME : AKULA VIVEK
ROLL NUMBER : 2403A51258
BATCH : 11

Subgroup B

B.1 — [S18B1] Apply surge/penalty rules (conditionals) Context:

Pricing in the real estate listings platform app uses a base per-km rate and time-based surge after peaks. Finance wants a deterministic calculator for receipts and audits.

Your Task:

Implement a fare function: fare = km * base_per_km * surgeMultiplier, where surge applies strictly after 18:00 local time.

Data & Edge Cases:

Input is a list of rides with `time` as HH:MM (24h) and `km` as float. Exactly 18:00 is non-surge; > 18:00 is surge.

AI Assistance Expectation:

Prompt AI to outline parsing HH:MM, applying conditionals, and rounding to 2 decimals; then implement and write a quick test.

Constraints & Notes:

No external libraries; round each fare to 2 decimals; do not mutate input.

Sample Input

[{'time': '07:45', 'km': 2.8}, {'time': '18:45', 'km': 6.2}]

Sample Output

[39.2, 108.5]

Acceptance Criteria: Correct surge threshold and rounding

Visual Studio Code:

```python
def calculate_fares(rides, base_per_km=14.0, surge_multiplier=1.25):
    fares = []
    for ride in rides:
        time_str = ride['time']
        km = ride['km']
        hour, minute = map(int, time_str.split(':'))
        # Surge applies strictly after 18:00
        if hour > 18 or (hour == 18 and minute > 0):
            multiplier = surge_multiplier
        else:
            multiplier = 1.0
        fare = km * base_per_km * multiplier
        fares.append(round(fare, 2))
    return fares

# Quick test
if __name__ == "__main__":
    sample_input = [
        {'time': '07:45', 'km': 2.8},
        {'time': '18:45', 'km': 6.2}
    ]
    print(calculate_fares(sample_input))  # Output: [39.2, 108.5]
```

Output :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  AUGMENT NEXT EDIT  CODE REFERENCE LOG

PS C:\Users\akula\OneDrive\Desktop\AIAC> & C:/Users/akula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/akula/OneDrive/Desktop/def calculate_fares(rides, base_per_km=1.py"
[39.2, 108.5]
```

Explanation :

Here's an explanation of the code:

- The function calculate_fares computes fares for a list of rides based on distance and time.
- Each ride is a dictionary with 'time' (in 'HH:MM' 24-hour format) and 'km' (distance as a float).
- The function parses the hour and minute from the time string.
- If the ride starts strictly after 18:00 (i.e., hour > 18 or hour == 18 and minute > 0), a surge multiplier is applied; otherwise, the base rate is used.
- The fare is calculated as km * base_per_km * multiplier and rounded to 2 decimal places.
- The function returns a list of fares, one for each ride.
- The test at the bottom demonstrates usage with sample input and prints the resulting fares.

B.2 — [S18B2] Debug rolling mean (o-by-one) Context:

A team in real estate listings platform noticed o -by-one bugs in a rolling KPI computation (moving averages) that undercount windows.

Your Task:

Use AI to identify the bug and fix the window iteration so all valid windows are included.

Data & Edge Cases:

For xs=[4, 5, 7, 10] and w=2, number of windows should be len(xs)-w+1.

AI Assistance Expectation:

Ask AI to add a failing test first, propose the minimal fix, and verify with the sample.

Constraints & Notes:

Guard invalid w (<=0 or >len(xs)); preserve O(n*w) simple solution.

Sample Input xs=[4, 5, 7, 10], w=2 Buggy code:

```
def rolling_mean(xs, w):     sums =
[]     for i in range(len(xs)-w):
window = xs[i:i+w]
sums.append(sum(window)/w)
return sums
```

Sample Output

[4.5, 6.0, 8.5]

Acceptance Criteria: All valid windows included; passes tests; no index errors

Visual studio code :

```python
# --- Python code for rolling mean with test and output ---

def rolling_mean(xs, w):
    if w <= 0 or w > len(xs):
        return []
    sums = []
    for i in range(len(xs)-w+1):
        window = xs[i:i+w]
        sums.append(sum(window)/w)
    return sums

# Sample input
xs = [4, 5, 7, 10]
w = 2

# Generate output
output = rolling_mean(xs, w)
print(output)  # Output: [4.5, 6.0, 8.5]
```

Output :

```
PS C:\Users\akula\OneDrive\Desktop\AIAC> & C:/Users/akula/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/a
[4.5, 6.0, 8.5]
```

Explanation :

- rolling_mean(xs, w):
  - Takes a list of numbers xs and a window size w. o If w is invalid (≤0 or larger than the list), returns an empty list.
  - For each valid window of size w in xs, it:
    - Slices the window: xs[i:i+w] Computes the mean: sum(window)/w Appends the mean to the result list.
  - Returns the list of rolling means.
- The sample input xs = [4, 5, 7, 10] and w = 2 produces the output [4.5, 6.0, 8.5], which are the means of [4,5], [5,7], and [7,10].
- The print(output) statement displays the result.