AI ASSISTED CODING

LAB EXAM

HALL TICKET : 2403A51270

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

CODE:

class FareCalculator:

    def \_\_init\_\_(self, base\_per\_km=7.0, surge\_multiplier=2.5):

        self.base\_per\_km = base\_per\_km

        self.surge\_multiplier = surge\_multiplier

    def calculate\_fares(self, rides):

        fares = []

        for ride in rides:

            hour, minute = map(int, ride['time'].split(':'))

            km = ride['km']

            if hour > 18 or (hour == 18 and minute > 0):

                multiplier = self.surge\_multiplier

            else:

                multiplier = 1.0

            fare = km \* self.base\_per\_km \* multiplier

            fares.append(round(fare, 2))

        return fares

sample\_input = [{'time': '07:45', 'km': 2.8}, {'time': '18:45', 'km': 6.2}]

calculator = FareCalculator()

print(calculator.calculate\_fares(sample\_input))  # Expected: [39.2, 108.5]

 OUTPUT:



B.2 — [S18B2] Debug rolling mean (off-by-one)

Context:

A team in real estate listings platform noticed off-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

CODE:

class RollingMeanCalculator:

    def \_\_init\_\_(self, data):

        self.data = data

    def compute(self, window\_size):

        if window\_size <= 0 or window\_size > len(self.data):

            return []

        means = []

        for i in range(len(self.data) - window\_size + 1):

            window = self.data[i:i + window\_size]

            means.append(sum(window) / window\_size)

        return means

xs = [4, 5, 7, 10]

w = 2

calculator = RollingMeanCalculator(xs)

print("Test 1:", calculator.compute(w))

OUTPUT:

