

AI-ASSISTED CODING

LAB TEST

SUBGROUP-B

H.NO: 2403A51236

NAME: S. SIRI

BATCH: 11

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: $\text{fare} = \text{km} * \text{base_per_km} * \text{surgeMultiplier}$, 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.

PROMPT:

Write a Python function `calculate_fares(rides)` where each ride is a dict with 'time' in HH:MM and 'km' as float.

- ➔ `base_per_km = 7.0`, `surge_multiplier = 2.5` applies strictly after 18:00. Round fares to 2 decimals and do not mutate input.

CODE AND OUTPUT:

```
B1.PY > ...  
1 def calculate_fares(rides):  
2     """  
3     Calculates fares for a list of rides.  
4     Each ride is a dict with 'time' (HH:MM) and 'km' (float).  
5     - base_per_km = 7.0  
6     - surge_multiplier = 2.5 applies strictly after 18:00  
7     Returns a new list of fares rounded to 2 decimals.  
8     """  
9     base_per_km = 7.0  
10    surge_multiplier = 2.5  
11    fares = []  
12    for ride in rides:  
13        time_str = ride['time']  
14        km = ride['km']  
15        hour, minute = map(int, time_str.split(":"))  
16        # Surge applies strictly after 18:00  
17        if hour > 18 or (hour == 18 and minute > 0):  
18            fare = km * base_per_km * surge_multiplier  
19        else:  
20            fare = km * base_per_km  
21        fares.append(round(fare, 2))  
22    return fares  
23  
24    # Example usage:  
25    rides = [{'time': '07:45', 'km': 2.8}, {'time': '18:45', 'km': 6.2}]  
26    print(calculate_fares(rides))  
27    # Expected output: [19.6, 108.5]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

cuments/AI_LAB_TEST/B1.PY
[19.6, 108.5]
PS C:\Users\siris\OneDrive\Documents\AI_LAB_TEST>

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 $\text{len}(xs)-w+1$.

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 $> \text{len}(xs)$); 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.

PROMPT:

The function below has an off-by-one bug in its rolling mean window iteration.

Write a failing test for `xs = [4,5,7,10]`, `w=2` expecting `[4.5,6.0,8.5]`,

then fix the function minimally so the test passes.


INITIAL CODE (BUGGY CODE) WITH FAILING TEST:

```
B2.py > rolling_mean
1 def rolling_mean(xs, w):
2     sums = []
3     for i in range(len(xs)-w): # Introduce off-by-one bug
4         window = xs[i:i+w]
5         sums.append(sum(window)/w)
6     return sums
7
8 # Test
9 xs = [4, 5, 7, 10]
10 w = 2
11 expected = [4.5, 6.0, 8.5]
12 result = rolling_mean(xs, w)
13 print("Result:", result)
14 assert result == expected, f"Expected {expected}, got {result}"
15
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + - [] ... [] x

```
PS C:\Users\siris\OneDrive\Documents\AI_LAB_TEST> & c:/Users/siris/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/siris/OneDrive/Documents/AI_LAB_TEST/B2.py
Result: [4.5, 6.0]
Traceback (most recent call last):
  File "c:\Users\siris\OneDrive\Documents\AI_LAB_TEST\B2.py", line 14, in <module>
    assert result == expected, f"Expected {expected}, got {result}"
           ^^^^^^^^^^^^^^^^^
AssertionError: Expected [4.5, 6.0, 8.5], got [4.5, 6.0]
PS C:\Users\siris\OneDrive\Documents\AI_LAB_TEST>
```

CORRECTED CODE WITH OUTPUT:



The screenshot shows a VS Code editor with a file named B2.py. The code defines a function `rolling_mean(xs, w)` that calculates a rolling mean. It includes a test case with `xs = [4, 5, 7, 10]` and `w = 2`, expecting the result `[4.5, 6.0, 8.5]`. The terminal at the bottom shows the command to run the script, which executed successfully, displaying the expected output.

```

Welcome
B1.py
B2.py
B2.py > rolling_mean
1 def rolling_mean(xs, w):
2     sums = []
3     for i in range(len(xs)-w+1): # Fixed off-by-one bug
4         window = xs[i:i+w]
5         sums.append(sum(window)/w)
6     return sums
7
8 # Test
9 xs = [4, 5, 7, 10]
10 w = 2
11 expected = [4.5, 6.0, 8.5]
12 result = rolling_mean(xs, w)
13 print("Result:", result)
14 assert result == expected, f"Expected {expected}, got {result}"
15
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python
PS C:\Users\siris\OneDrive\Documents\AI_LAB_TEST> & C:/Users/siris/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/siris/OneDrive/Do
cuments/AI_LAB_TEST/B2.py
Result: [4.5, 6.0, 8.5]
PS C:\Users\siris\OneDrive\Documents\AI_LAB_TEST>

```